Implementation of Bourbaki's Elements of Mathematics in Coq: Part One, Theory of Sets

José Grimm

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Implementation of Bourbaki's Elements of Mathematics in Coq: Part One Theory of Sets

José Grimm

N° 6999 — version 3

initial version July 2009 — revised version March 2010

Algorithmics, Programming, Software and Architecture

ISSN 0249-6399 ISRN INRIA/RR--6999--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

Implementation of Bourbaki's Elements of Mathematics in Coq: Part One Theory of Sets

José Grimm*

Domain : Algorithmics, Programming, Software and Architecture

Équipe-Projet Apics

Rapport de recherche n° 6999 — version 3 — initial version July 2009 — revised version March 2010 — 209 pages

Abstract: We believe that it is possible to put the whole work of Bourbaki into a computer. One of the objectives of the Gaia project concerns homological algebra (theory as well as algorithms); in a first step we want to implement all nine chapters of the book Algebra. But this requires a theory of sets (with axiom of choice, etc.) more powerful than what is provided by Ensembles; we have chosen the work of Carlos Simpson as basis. This reports lists and comments all definitions and theorems of the Chapter "Theory of Sets". The code (including almost all exercises) is available on the Web, under http://www-sop.inria.fr/apics/gaia.

Version one was released in July 2009, version 2 in December 2009, version 3 in March 2010. There are small differences, marked in footnotes.

Key-words: Gaia, Coq, Bourbaki, Formal Mathematics, Proofs, Sets

Work done in collaboration with Alban Quadrat, based on previous work of Carlos Simpson (CNRS, University of Nice-Sophia Antipolis)

* Email: Jose.Grimm@sophia.inria.fr

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Implémentation des Éléments de mathématiques de Bourbaki en Coq, partie 1 Théorie des ensembles

Résumé : Nous pensons qu'il est possible de mettre dans un ordinateur l'ensemble de l'œuvre de Bourbaki. L'un des objectifs du projet Gaia concerne l'algèbre homologique (théorie et algorithmes); dans une première étape nous voulons implémenter les neuf chapitres du livre Algèbre. Au préalable, il faut implémenter la théorie des ensembles. Nous utilisons l'Assistant de Preuve Coq; les choix fondamentaux et axiomes sont ceux proposées par Carlos Simpson. Ce rapport liste et commente toutes les définitions et théorèmes du Chapitre théorie des ensembles. Presque tous les exercises ont été résolus. Le code est disponible sur le site Web http://www-sop.inria.fr/apics/gaia.

Mots-clés : Gaia, Coq, Bourbaki, mathématiques formelles, preuves, ensembles

Chapter 1

Introduction

1.1 Objectives

Our objective (it will be called the *Bourbaki Project* in what follows) is to show that it is possible to implement the work of N. Bourbaki, "Éléments de Mathématiques"[3], into a computer, and we have chosen the Coq Proof Assistant, see [4, 1]. All references are given to the English version "Elements of Mathematics"[2], which is a translation of the French version (the only major difference is that Bourbaki uses an axiom for the ordered pair in the English version and a theorem in the French one). We start with the first book: theory of sets. It is divided into four chapters, the first one describes formal mathematics (logical connectors, quantifiers, axioms, theorems). Chapters II and III form the basis of the theory; they define sets, unions, intersections, functions, products, equivalences, orders, integers, cardinals, limits. The last chapter describes structures.

An example of structure is the notion of real vector space: it is defined on a set E, uses the set $\mathbb{R}$ of real numbers as auxiliary set, has some characterization (there are two laws on E, a zero, and a action of $\mathbb{R}$ over E), and has an axiom (the properties of the the laws, the action, the zero, etc.). A complete example of a structure is the *order*; given a set A, we have as characterization $s \in \mathfrak{P}(A \times A)$ and the axiom "$s \circ s = s$ and $s \cap s^{-1} = \Delta_A$". We shall see in the second part of this report that an ordering satisfies this axiom, but it not clear if this kind of construction is adapted to more complicated structure (for instance a left module on a ring). Given two sets A and A′, with orderings s and s', we can define $\sigma(A, A', s, s')$, the set of increasing functions from A to A′. An element of this set is called a σ-morphism. In our implementation, the "set of functions f such that ..." does not exists; we may consider the set of graphs of functions (this is well-defined), but we can also take another position: we really need σ to be a set if we try to do non-trivial set operations on it, for instance if we want to define a bijection between σ and σ′; these are non-obvious problems, dealt with by the theory of categories. There is however another practical problem; Bourbaki very often says: let E be an ordered set; this is a short-hand for a pair (A, s). Consider now a monoid $(A, +)$. Constructing an ordered monoid is trivial: the characterization is the product of the characterizations, and the axiom is the conjunction of the axioms. The ordered monoid could be $(A, (s, +))$. If f is a morphism for s, and $u \in A$, then the mapping $x \mapsto f(x + u)$ is a morphism for s, provided that + is compatible with s. If we want to convert this into a theorem in Coq, the easiest solution is to define an object X equivalent to $(A, (s, +))$, a way to extract $X' = (A, s)$ and $X'' = (A, +)$ from X, an operation s on A obtained from X or X′, and change the definition of σ: it should depend on X′ rather than on A and s. The compatibility condition is then a property of X, $\sigma(X, Y)$ and $\sigma(X', Y)$ are essentially the same objects, if $f \in \sigma(X, Y)$ we can con-

sider $f' = x \mapsto f(x+u)$, and show $f' \in \sigma(X,Y)$. From this we can deduce the mapping from $\sigma(X',Y)$ into $\sigma(X',Y)$ associated to $f \mapsto f'$.

1.2 Background

We started with the work of Carlos Simpson[1], who has implemented the Gabriel-Zisman localization of categories in a sequence of files: *set.v*, *func.v*, *ord.v*, *comb.v*, *cat.v*, and *gz.v*. Only the first three files in this list are useful for our project. The file *ord.z* contains a lot of interesting material, but if we want to closely follow Bourbaki, it is better to restart everything from scratch. The file *func.v* contains a lot of interesting constructions and theorems, that can be useful when dealing with categories. For instance, it allows us to define morphisms on the category of left modules over a ring. The previous discussion about structures and morphism explains why only half of this file is used.

This report is divided in two parts. The first part deals with implementation of Chapter II, "Theory of sets", and the second part with chapter III, "Ordered sets, cardinals; integers" of [2] Each of the six sections of Bourbaki gives a chapter in this report (we use the same titles as in Bourbaki) but we start with the description of the two files *set.v* and *func.v* by Carlos Simpson (it is a sequence of modules). Their content covers most of Sections 1 and 2 ("Collectivizing relations" and "Ordered pairs").

1.3 Notations

Choosing tractable notations is a difficult task. We would like to follow the definitions of Bourbaki as closely as possible. For instance he defines the union of a family $(X_\iota)_{\iota\in I}(X_\iota \in \mathfrak{G})$. Classic French typography uses italic lower-case letters, and upright upper-case letters, but the current math tradition is to use italics for both upper- and lower-case letters for variables; constants like pr_1 and Card use upright font. The set of integers is sometimes noted $\mathbb{N}$; but Bourbaki uses only **N**. Some characters may have variants (for instance, the previous formula contains a Fraktur variant of the letter G). In the XML version of this document we do not use the Unicode character U+1D50A (because not most browsers do not have the glyph), but a character with variant, so that there is little difference between G, **G**, G, *G*, $\mathbb{G}$, $\mathfrak{G}$. In this document we use only one variant of the Greek alphabet (Unicode provides normal, italic, bold, bold-italic, sans-serif and sans-serif bold italic; as a consequence, the XML version shows generally a slanted version of Greek characters, where the Pdf document uses an upright font).

We can easily replace lower Greek letters by their Latin equivalents (there is little difference between $(X_\iota)_{\iota\in I}$ and $(X_i)_{i\in I}$). We can replace these unreadable old German letters by more significant ones. We must also replace I by something else, because this is a reserved keyword in Coq (and *in* is reserved too). In the original version, C. Simpson reserved the letters A, B and E. Thus, a phrase like: let A and B be two subsets of a set E, and I = A × B, all four identifiers are reserved letters in Simpson's framework. Note that, traditionally, French mathematicians use roman upright upper case letters and italics lower case letters for variables;

Quantities named R, B, X, Y, and Z by Simpson have been renamed to Ro, Bo, Xo, Yo and Zo. Quantity A has been removed (it was a prefix version of &). Quantity E has been renamed

[1] http://math.unice.fr/~carlos/themes/verif.html

Bset then Set: this is the type of a Bourbaki set. It will still be denoted by $\mathscr{E}$ here. In our framework, the reserved single-letter identifiers are I J L O P Q S V W.

Coq reserves the letter I as a proof of *True*, the letter O as the integer 0 and the letter S for the function $n \mapsto n+1$ on integers. An ordered pair with values x and y is a term z that has two projections $\mathrm{pr}_1 z = x$ and $\mathrm{pr}_2 z = y$. The constructor is called *bpair*[2] in Coq, and the destructors are called *pr1* and *pr2*. We shall reserve the letters J for the constructor and P, Q for the destructors, so that *J (P z) (Q z) = z* for all pairs z (see section 2.6 for details).

Bourbaki has a section titled "definition of a function by means of a term". An example would be $x \mapsto (x,x)$ $(x \in \mathbb{N})$. This corresponds to the Coq expression *fun x:nat => (x,x)*. According to the Coq documentation, the expression "defines the abstraction of the variable *x*, of type *nat*, over the term *(x,x)*. It denotes a function of the variable *x* that evaluates to the expression *(x,x)*". Bourbaki says "a mapping of A into B is a function f whose source is equal to A and whose target is equal to B". The distinction between the terms function and mapping is subtle: there is a section called "sets of mappings of one set into another"; it could have been: "sets of functions whose source is equal to some given set and whose target is equal to some other given set". It is interesting to note that the term 'function' is used only once in the exercises to Chapter III, in a case where 'mapping' cannot be used because Bourbaki does not specify the set B.

In what follows, we shall use the term 'function' indifferently for S, or the mapping $n \mapsto n+1$, or the abstraction *n => S n*. Given a set A, we can consider the graph g of this mapping when n is restricted to A. This construction is so important that we reserve the letter L for it. Given a set B, if our mapping sends A to B, we can consider the (formal) function f associated to the mapping with source A and target B. We shall denote this by BL. These two objects f and g have the important property that, if n is in A, there is an m denoted by $f(n)$ or $g(n)$ such that $m = n+1$ (we have the additional property that $f(n)$ is in B). A short notation is required for the mapping from $(g,n) \mapsto g(n)$, or $(f,n) \mapsto f(n)$. We shall use the letters V and W respectively. In this document, we shall use standard notations, for instance pr_1 and pr_2 for P and Q, when they exist, calligraphic letters like $\mathscr{V}$ or $\mathscr{W}$ for some objects like V and W, and a slanted font like *is_function* for the general case. Note that *J x y* is a Coq expression meaning the application of J to both arguments x and y.

There a possibility to change the Coq parser and pretty printer so that `(x,y)` is read as `pair x y`, and `{ x : A | P }` is read as the set of all x in A satisfying the predicate P. We shall not use this feature here. In fact, these are standard notations in Coq for notions that are related but not exactly identical to ours.

1.4 Description of formal mathematics

Terms and relations. A mathematical theory $\mathscr{T}$ is a collection of words over a finite alphabet formed of letters, logical signs and specific signs. Logical signs are $\square$, τ, $\vee$, $\neg$ (the first two signs are specific to Bourbaki, the other ones, disjunction and negation, have their usual meaning). Specific signs are $=$, $\in$, letters are x, y, A, A′, A″, A‴, and "at any place in the text it is possible to introduce letters other than those which have appeared in previous arguments" [2, p. 15] (any number of prime signs is allowed; this is not in contradiction with the finiteness of the alphabet). An assembly is a sequence of signs and links. Some assemblies are well-formed according to some grammar rules. In Backus-Naur form they are:

Term := letter | τ_{letter} (Relation) | Ssign Term_1 ... Term_n

[2] This is called 'pair' in Simpson and in version 1 of this report

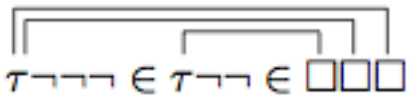
Relation := ¬ Relation | ∨ Relation Relation | Rsign Term$_1$ ... Term$_n$
Each sign has to be followed by the appropriate number of terms: □ takes none, ∈ and = are followed by two signs, and one can extend Bourbaki to non-standard analysis [7] by introducing a specific sign $^{\text{st}}$ of weight 1 qualifying the relation that follows to be standard. Each sign is substantific as □ (it yields a term) or relational as = (it yields a relation).

We shall see below that $\tau_x(\text{R})$ has to be interpreted as the expression where all occurrences of x in R are replaced by □ and linked to the τ. Parentheses are removed. This has one advantage: there is no x in $\tau_x(\text{R})$, hence substitution rules become trivial. For instance, the function $x \mapsto x+y$ is constructed by using τ, it is identical to the function $z \mapsto z+y$. If we want to replace y by z, we get $x \mapsto x+z$, but not $z \mapsto z+z$. In Coq, the variable y appears *free* in $x \mapsto x+y$, and the variable x appears *bound* in the same expression. Renaming bound variables is called α-*conversion*. Two α-convertible terms are considered equal in Coq.

The Appendix to Chapter I describes an algorithm that decides whether an assembly is a term, a relation, or is ill-formed. It works in two stages. In the first stage, links are ignored. A classical result in computer science is that there exists a program (called a *parser*) that recognizes all *significant words* (i.e., well-formed assemblies without links). We can associate a number to each sign (for instance 262 to 'a', 111 to '=') and thus to each assembly (for instance, 262111262 to 'a=a'). This will be called the Gödel number of the assembly, see [5] for an example. Two distinct assemblies have distinct Gödel numbers. The set of Gödel numbers is a recursively enumerable set. Given assemblies A_1, A_2, A_3, etc, one can form the concatenation $\text{A}_1\text{A}_2\text{A}_3\ldots$. If each assembly is a significant word, there is a unique way to recover A_i from the concatenation, hence from the Gödel number of the concatenation.

A *demonstrative text* for Bourbaki is a sequence of assemblies $\text{A}_1\text{A}_2\ldots\text{A}_n$, that contains a *proof*, which is a subsequence $\text{A}'_1\text{A}'_2\ldots\text{A}'_m$ of relations, where each A'_i can be shown to be true by application of a basic derivation rule that uses only A'_j for $j<i$. Each A'_i is a *theorem*. We shall use a variant: a *proof-pair* is a sequence of relations $\text{A}'_1\text{A}'_2\ldots\text{A}'_m$ satisfying the same conditions as above, and a *theorem* is the last relation A'_m in a proof-pair. If our basic rules are simple enough, the property of a number g to be the Gödel number of a proof-pair is primitive recursive. From this, one can deduce the existence of a true statement that has no proof (this is Gödel's Theorem).

An assembly A containing links is analyzed by using *antecedents*, which are assemblies of the form $\tau_x(\text{R})$ (where x is some variable) that are identical to A if x is substituted in R and links are added. The algorithm for deciding that an assembly with links is a term or a relation is rather complicated. Bourbaki gives three examples of assemblies with links; the antecedent of the first one is $\tau_x(x\in y)$ (there is a single link); the antecedent of the second one is $\tau_x(x\in\text{A}' \implies x\in\text{A}'')$ (there are two links); the third one is the empty set, see picture below. One can replace these links by the De Bruijn indices, so that the empty set would become $\tau\neg\neg\neg\in\tau\neg\neg\in 121$. This has two drawbacks: the first one is that 121 could be understood as one integer or a sequence of three integers, the second is that this notation assumes that integers are already defined. The remedy to the first problem would be to insert a separator (for instance a square) and a remedy to the second would be to use a base-one representation of integers; the empty set would be $\tau\neg\neg\neg\in\tau\neg\neg\in\square-\square--\square-$. The scope of the second τ is the scope of its operator, thus $\neg\neg\in\square\square$. This means that the two squares are in the scope of both τ, are are linked to the second and first τ respectively. The third square is in the scope of the first τ only, hence is linked to the first τ. Formal mathematics in Bourbaki is so complicated that the □ symbol is, in reality, never used.



Denote by (***B***|***x***) ***A*** the assembly obtained by replacing ***x***, wherever it occurs in ***A***, by the assembly ***B***. Bourbaki has some criteria of substitutions, CS1, CS2, etc, that are rules about substitutions. For instance CS3 says that $\tau_{\boldsymbol{x}}(\boldsymbol{A})$ and $\tau_{\boldsymbol{x}'}(\boldsymbol{A}')$ are identical if $\boldsymbol{A}'$ is $(\boldsymbol{x}'|\boldsymbol{x})$ ***A*** provided that $\boldsymbol{x}'$ does not appear in ***A*** (informally: since x does not appear in τ_x(A), the name of the variable x is irrelevant). Formative criteria CF1, CF2, etc., give rules about well-formedness of assemblies. For instance CF8 says that (***T***|***x***) ***A*** is a term or a relation whenever ***A*** is a term or a relation, ***T*** is a term, ***x*** is a letter.

Abbreviations are allowed, so that $\vee\neg$ can be replaced by $\Longrightarrow$, and $\neg\in$ can be replaced by $\notin$. Abbreviations may take arguments, for instance $\wedge$AB is the same as $\neg\vee\neg$A$\neg$B. A term may appear more than once, for instance $\Longleftrightarrow$ AB is the same as $\wedge\Longrightarrow$ AB $\Longrightarrow$ BA, and after expansion $\neg\vee\neg\vee\neg$AB$\neg\vee\neg$BA. The logical connectors $\neg$, $\vee$ and $\wedge$ are written ~, \/, and /\ in Coq (we shall use & instead of $\wedge$, since it is easier to type). Note that in Coq, A $\rightarrow$ B is the type of a function from A to B but also means A $\Longrightarrow$ B. There is no limit on the number of abbreviations (Bourbaki invented $\emptyset$ as a variant of ø). Unicode provides a lot of symbols, but few of them are available in LaTeX or in Web browsers.

Starting with Section 2, Bourbaki switches to infix notation. For instance, whenever ***A*** and ***B*** are relations so is $\vee\neg\neg\vee\neg\boldsymbol{A}\neg\boldsymbol{BA}$, by virtue of CF5 and CF9. Using abbreviations, this relation can be written as $\Longrightarrow\wedge\boldsymbol{ABA}$. The infix version is (***A*** and ***B***) $\Longrightarrow$ ***A***. In order to remove ambiguities, parentheses are required, but Bourbaki says: "Sometimes we shall leave out the brackets" [2, p. 24], in the example above three pairs of brackets are left out. In some cases Bourbaki writes A $\cup$ B $\cup$ C. This can be interpreted as (A $\cup$ B) $\cup$ C or A $\cup$ (B $\cup$ C). These are two distinct objects that happen to be equal: formally, the relation (A $\cup$ B) $\cup$ C = A $\cup$ (B $\cup$ C) is true. Similarly A $\vee$ B $\vee$ C is ambiguous, but it happens, according to C24, that (A $\vee$ B) $\vee$ C and A $\vee$ (B $\vee$ C) are equivalent (formally: related by $\Longleftrightarrow$). In Coq, we use *union2* as prefix notation for $\cup$, so we must chose between $\cup$($\cup$AB)C or $\cup$A($\cup$BC). Function calls are left-associative, and brackets are required where indicated. We use \/ as infix notation for $\vee$, parentheses may be omitted, the operator is right associative.

Theorems and proofs. Each relation can be true or false. To say that P is false is the same as to say that $\neg$P is true. To say that P is either true or false is to say that P $\vee\neg$P is true. A relation is true by assumption or deduction. A relation can be both true and false, case where the current theory is called contradictory (and useless, since every property is then true). There may be relations P for which it is impossible to deduce that P is true and it is also impossible to deduce that P is false (Gödel's theorem). A property can be independent of the assumptions. This means that it is impossible to deduce P or $\neg$P; in other words, adding P or $\neg$P does not make the theory contradictory. An example is the axiom of foundation (see below), or the continuum hypothesis (every uncountable set contains a subset which has the power of the continuum).

Some relations are true by assumption; these are called axioms. An axiom scheme is a rule that produces axioms. The list of axioms and schemes used by Bourbaki are given at the end of the document. A true relation is called a Theorem (or Proposition, Lemma, Remark, etc). A conjecture is a relation believed to be true, for which no proof is currently found. As said above, in Bourbaki, a theorem is a relation with a proof, which consists of a sequence of true statements, the theorem is one of them, and each statement R in the sequence is either

an axiom, follows by applications of rules (the axiom schemes) to previous statements, or there are two previous statements S and T before R, where T has the form S $\Longrightarrow$ R.

It is very easy for a computer to check that an annotated proof is correct (provided that we use a parsable syntax); but a formal proof is in general huge. Example of formal proofs can be found in [5]; the theory used there is simpler than Bourbaki's, but contains arithmetics on integers. We give here a proof of 1 + 1 = 2:

(1)	∀a:∀b:(a+Sb)=S(a+b)	axiom 3
(2)	∀b:(S0+Sb)=S(S0+b)	specification (S0 for a)
(3)	(S0+S0)=S(S0+0)	specification (0 for b)
(4)	∀a:(a+0)=a	axiom 2
(5)	(S0+0)=S0	specification (S0 for a)
(6)	S(S0+0)=SS0	add S
(7)	(S0+S0)=SS0	transitivity (lines 3,6)

The proof is formed of the statements in the second column; the annotations of the third column are not part of the formal proof. The line numbers can be used in the annotations. In Coq, the annotations are part of the proof. The principle is: a theorem is a function and applying the theorem means applying the function. For instance, transitivity of equality is a function *trans_eq*; in line (7) we apply it to two arguments, the statements of lines 3 and 6. The statement of line 6 is obtained by applying *f_equal* with argument S to the statement that precedes (the *f_equal* theorem states that for every function f and equality $a = b$ we have $f(a) = f(b)$). In Coq, a proof is a tree, the advantage is that we do not need to worry about line numbers.

Bourbaki has over 60 criteria that help proving theorems. The first one says: if $\boldsymbol{A}$ and $\boldsymbol{A} \Longrightarrow \boldsymbol{B}$ are theorems, then $\boldsymbol{B}$ is a theorem. This is not a theorem, because it requires the fact that $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations. On the other hand $x = x$ is a theorem (the first in the book). The difference is the following: if A and B are letters then A $\Longrightarrow$ B is not well-formed. Until the end of E.II.5, Bourbaki uses a special font as in $\boldsymbol{A} \Longrightarrow \boldsymbol{B}$ to emphasize that $\boldsymbol{A}$ and $\boldsymbol{B}$ are to be replaced by something else.

Criterion C1 works as follows. If $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n$ and $\mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m$ are two proofs, if the first one contains A, if the second one contains A $\Longrightarrow$ B, then

$$\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{B}$$

is a proof that contains B. Assume that we have two annotated proofs R_i and S_j, where A is R_n and A $\Longrightarrow$ B is S_m. Each statement has a line number, and we can change these numbers so that they are all different (this is a kind of α-conversion). Let N and M be the line numbers of R_n and S_m. We get an annotated proof by choosing a line number for the last statement, and annotating it by: detachment N M (this is also known as syllogism, or Modus Ponens).

Criterion C6 says the following: assume $\boldsymbol{P} \Longrightarrow \boldsymbol{Q}$ and $\boldsymbol{Q} \Longrightarrow \boldsymbol{R}$. From axiom scheme S4, we get $(\boldsymbol{Q} \Longrightarrow \boldsymbol{R}) \Longrightarrow ((\boldsymbol{P} \Longrightarrow \boldsymbol{Q}) \Longrightarrow (\boldsymbol{P} \Longrightarrow \boldsymbol{R}))$. Applying Criterion C1 gives $(\boldsymbol{P} \Longrightarrow \boldsymbol{Q}) \Longrightarrow (\boldsymbol{P} \Longrightarrow \boldsymbol{R})$. Applying it again gives $\boldsymbol{P} \Longrightarrow \boldsymbol{R}$. If $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n$ and $\mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m$ are proofs of $\boldsymbol{P} \Longrightarrow \boldsymbol{Q}$ and $\boldsymbol{Q} \Longrightarrow \boldsymbol{R}$ then a proof of $\boldsymbol{P} \Longrightarrow \boldsymbol{R}$ is

$$\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{A}_4, \mathrm{D}_y, \mathrm{D}_y.$$

Here A_4 and D_y are to be replaced by the appropriate relation, or in the case of an annotated proof, by the appropriate annotation (for instance in the case of A_4, we must give the values of three arguments of the axiom scheme S4, in the case of detachment D_y we must give the position of the arguments of the syllogism in the proof tree).

Criterion C8 says $\boldsymbol{A} \implies \boldsymbol{A}$. This is a trivial consequence from S2, $\boldsymbol{A} \implies (\boldsymbol{A} \vee \boldsymbol{A})$ and S1, $(\boldsymbol{A} \vee \boldsymbol{A}) \implies \boldsymbol{A}$. This is by definition $\neg \boldsymbol{A} \vee \boldsymbol{A}$, and is called the "Law of Excluded Middle".

There is a converse to C6. If we can deduce, from the statement that A is true, a proof of B, then A $\implies$ B is true. This is called *method of the auxiliary hypothesis*. Almost all theorems we shall prove in Coq have this form.

Criterion C21 says that $\vee\neg\neg\vee\neg\boldsymbol{A}\neg\boldsymbol{B}\boldsymbol{A}$ is a theorem, whenever $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations. We have already seen this assembly and showed that it is a relation. If we could quantify relations, the criterion could be converted into a theorem that says "(∀A)(∀B)((A and B) $\implies$ A)". In Coq, we can quantify everything and our theorem is then:

```
example =
fun (A B : Prop) (H : A /\ B) => and_ind (fun (H0 : A) (_ : B) => H0) H
     : forall A B : Prop, A /\ B -> A
```

This has the form "name = proof : value". The last line is the value of the theorem. The second line is the proof. As you can see, this is not just a sequence of statements with their justification, but function calls. It applies *and_ind* to f_2 and H where f_2 is a function of two arguments that returns the first one, and ignores the second (the proof of B). We show here the function:

```
and_ind =
fun A B P : Prop => and_rect (A:=A) (B:=B) (P:=P)
     : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P
Arguments A, B, P are implicit
```

According to this definition, *and_ind* takes three implicit arguments, named A, B and P, its value is a function that takes two arguments, a function f_2 and a proof of A ∧ B. Here f_2 is a function that maps A and B to P. The result is a proof of P. Arguments A, B and P are deduced from f_2 and need not be given (on the second line you see expressions of the form `(A:=A)`, this is because *and_rect* has three implicit arguments, that must be explicitly given). With our function f_2, P is the first argument hence A. As you can see, proofs in Coq are often hard to read. The important point is that Coq is a proof assistant: it is a system in which inventing a proof is easy. Here in an example.

```
Lemma example: forall A B, A /\ B -> A. intros. induction H. exact H. Qed.
```

The proof is formed of three parts. We first say *intros*. This means: consider an environment in which A and B are propositions (very often, types are guessed by Coq), and A ∧ B is true; we must show A. In this environment, everything has a name and H is the name of the assumption A ∧ B. Then we say *induction H*. This has as effect to remove H and replace it by all possible consequences (well, not *all*, since *and_ind* shows that there is one for each function f_2). The consequences are A and B, they are named as H and H0. The last step is *exact H*. This means that our goal is true, since it is the assumption H. We say *Qed* when the proof is complete.

If P and Q are propositions, one can show that ¬¬P ⇒ P, ((P ⇒ Q) ⇒ P) ⇒ P, P ∨ ¬P, ¬(¬P ∧ ¬Q) ⇒ P ∨ Q and (P ⇒ Q) ⇒ (¬P ∨ Q) are equivalent. These statements are unprovable in Coq. They are true in Bourbaki since the last statement is a tautology. In [5], there is the Double-tilde Rule that says that the string '~~' can be deleted from any theorem, and can be inserted into any theorem provided that the resulting string is itself well-formed. We solve this problem by adding the first statement as axiom. Then all theorems of Bourbaki

can be proved in Coq. There are still two difficulties: the first one concerns the status of τ (see below); the second concerns sets. Bourbaki says *in the formalistic interpretation of what follows, the word "set" is to be considered as strictly synonymous with "term"* [2, p. 65]. Recall that there are only two kinds of valid assemblies, namely terms and relations. We shall see in the next chapter how to implement sets in Coq. The standard library of Coq contains one implementation *Ensembles* in which a set is a relation. Our work matches Bourbaki better.

Quantified theories Bourbaki defines $\tau_x(\mathrm{R})$ as the construction obtained by replacing all x in R by $\square$, adding τ in front, and drawing a line between τ and this square. For instance $\tau\neg\neg\neg\in\tau\neg\neg\in\square\square\square$ with links as shown on the previous figure. It corresponds to: $\tau_x(\neg\neg\neg\in\tau_y(\neg\neg\in yx)x)$. The positions of the parentheses is fixed by the structure: $\neg$ takes one argument, while $\in$ takes two arguments. For the expression to make sense it is necessary that all $\square$ related to τ_y are in scope of the parentheses. If we admit that the double negation of P is P and use infix notation, the previous term is equivalent to $\tau_x(\tau_y(y\in x)\notin x)$. This is the empty set.

Denote by $(\mathrm{T}|x)$ R the expression R where all free occurrences of the letter x have been replaced by the term T. Paragraph 2.4.1 of [1] explains that this is a natural operation in Coq; the right amount of α-conversions are done so that free occurrences of variables in T are still free in all copies of T. For instance, if R is $(\exists z)(z=x)$, if we replace x by z, the result becomes $(\exists w)(w=z)$. These conversions are not needed in Bourbaki: there is no x in $\tau_x(\mathrm{R})$ and no z in $(\exists z)(z=x)$. Of course, if we want to simplify $(z|x)\ (\exists z)(z=x)$, we can replace it by $(z|x)\ (\exists w)(w=x)$ (thanks to rule CS8) then by $(\exists w)((z|x)\ (w=x))$ (thanks to rule CS9), then simplify as $(\exists w)(w=z)$.

Bourbaki defines $(\forall \boldsymbol{x})\boldsymbol{R}$ as "not $((\exists \boldsymbol{x})$ not $\boldsymbol{R})$", whereas *forall x:T, R* is a Coq primitive, whose meaning is (generally) obvious; instead of T, any type can be given, it may be omitted if it is deducible via type inference. The expression $(\forall_{\boldsymbol{T}}\boldsymbol{x})\boldsymbol{R}$ is defined in Bourbaki, similar to the Coq expression, but not used later on; we shall not use it here. The dual expression *exists x:T, R* is equivalent in Coq to *ex(fun x:T=>R)*. Here, *ex* is an inductive object. If P is the argument, and if for some witness x, $\mathrm{P}(x)$ is true, then *ex P* is defined. From this, we deduce that for some x, $\mathrm{P}(x)$ is true. Assume that $f:\mathrm{A}\to\mathrm{B}$ is a surjective function. This means that for each y in B there is an x in A that f maps to y. This does not mean that there is a g such that for all y, $f(g(y))=y$. The existence of g is related to the "axiom of choice".

Bourbaki defines $(\exists \boldsymbol{x})\boldsymbol{R}$ as $(\tau_{\boldsymbol{x}}(\boldsymbol{R})|\boldsymbol{x})\boldsymbol{R}$. Write y instead of $\tau_{\boldsymbol{x}}(\boldsymbol{R})$. Our expression is $(y|\boldsymbol{x})\boldsymbol{R}$. It does not contain the variable $\boldsymbol{x}$, since $\boldsymbol{x}$ is not in y. If $(\exists \boldsymbol{x})\boldsymbol{R}$ is true, then $\boldsymbol{R}$ is true for at least one object, namely y. This object is explicit: we do not need to introduce a specific axiom of choice. Axiom scheme S5 states the converse: if for some $\boldsymbol{T}$, $(\boldsymbol{T}|\boldsymbol{x})\ \boldsymbol{R}$ is true, then $(\exists \boldsymbol{x})\boldsymbol{R}$ is true.

Let's give an example of a non-trivial rule. As noted in [5], it is possible to show, for each integer n, that $0+n=n$ (where addition is defined by $n+0=n$ and $n+\mathrm{S}m=\mathrm{S}(n+m)$), but it is impossible to prove $\forall n, 0+n=n$. The following induction principle is thus introduced: "Suppose u is a variable, and $\mathrm{X}\{u\}$ is a well-formed formula in which u occurs free. If both $\forall u:\langle \mathrm{X}\{u\}\supset \mathrm{X}\{\mathrm{S}u/u\}\rangle$ and $\mathrm{X}\{0/u\}$ are theorems, then $\forall u:\mathrm{X}\{u\}$ is also a theorem."

Criterion C61 [2, p. 168] is the following: Let $\mathrm{R}\{n\}$ be a relation in a theory $\mathscr{T}$ (where n is not a constant of $\mathscr{T}$). Suppose that the relation

$$\mathrm{R}\{0\} \text{ and } (\forall n)((n \text{ is an integer and } \mathrm{R}\{n\}) \implies \mathrm{R}\{n+1\})$$

is a theorem of $\mathscr{T}$. Under these conditions the relation

$$(\forall n)((n \text{ is an integer}) \implies \mathrm{R}\{n\}).$$

is a theorem of $\mathscr{T}$.

The syntax is different, but the meaning is the same. This criterion is a consequence of the fact that a non-empty set of integers is well-ordered. In Coq, a consequence of the definition of integers is the following induction principle:

```
nat_ind =
fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Equality In Bourbaki, equality is defined by the two axioms schemes S6 and S7 (see section 9.1 for details). The first one says says that if P is a property depending on a variable z, if $x = y$, then $\mathrm{P}(x)$ and $\mathrm{P}(y)$ are equivalent. The second axiom says that if Q and R are two properties depending on a variable z, if $\mathrm{Q}(z)$ and $\mathrm{R}(z)$ are equivalent for all z, then $\tau_z(\mathrm{Q}) = \tau_z(\mathrm{R})$.

Assume that we want to show $1+1=2$. How can we proceed? Both objects $1+1$ and 2 are sets, i.e. terms, so by definition are of the form $\tau_z(\mathrm{Q})$ and $\tau_z(\mathrm{R})$. It suffices to prove that Q and R are equivalent. Is it necessary? Let $a = \tau_z(z \in \{1,2,3\})$ and $b = \tau_z(z \in \{1,2,4\})$. We have two non-equivalent propositions, so that it is not possible to prove $a = b$. Let's add this as an axiom. Then we have $a \in \{1,2\}$. Proof. Let $\mathrm{A} = \{1,2,3\}$ and $\mathrm{B} = \{1,2,4\}$. Since A and B are non-empty sets, we have $a \in \mathrm{A}$ and $b \in \mathrm{B}$. Moreover $\mathrm{A} \cap \mathrm{B} = \{1,2\}$. The claim is now: if $a \in \mathrm{A}$ and $b \in \mathrm{B}$ and $a = b$, then $a \in \mathrm{A} \cap \mathrm{B}$. More basically, the claim is: if $a = b$, if $a = 1$ or $a = 2$ or $a = 3$, and if moreover $b = 1$ or $b = 2$ or $b = 4$, then $a = 1$ or $a = 2$. Neither of the two statements $a = 1$ nor $a = 2$ is provable. However $a = 4$ and $b = 3$ are false, thus $a = 3$ is false, thus the conclusion.

If objects x are y are defined by a functional property (see below), then $x = y$ is equivalent to $\mathrm{Q} \iff \mathrm{R}$. This is not the case of cardinals. In fact, they are defined via a variable x and a fixed property P. We have $1+1 = \tau_z(\mathrm{P}(z,x))$ and $2 = \tau_z(\mathrm{P}(z,x'))$. The property P is an equivalence relation, so that $1+1=2$ is equivalent to $\mathrm{P}(x,x')$. In these two cases, comparing two sets is the same as comparing two relations.

In our framework, few objects are defined via τ, and Axiom Scheme S7 is rarely used. For instance $\{1+1\} = \{2\}$ is a trivial consequence of $1+1=2$, and Criterion C44. Let's prove this criterion and the first three theorems of Bourbaki.

Theorem 1 is $x = x$. Bourbaki uses an auxiliary theory in which x is not a constant, so that $(\forall x)(\boldsymbol{R} \iff \boldsymbol{R})$ is true, whatever the relation $\boldsymbol{R}$. Note that x is a letter, thus a term, and it could denote the set of real numbers $\mathbf{R}$, case where quantification over x makes no sense, while $\boldsymbol{R}$ is just a notation. Scheme S7 gives $\tau_x(\boldsymbol{R}) = \tau_x(\boldsymbol{R})$. This can be rewritten as $(\tau_x \boldsymbol{R}|x)(x = x)$. Let $\boldsymbol{S}$ denote $x = x$ and $\boldsymbol{R}$ denote $\neg \boldsymbol{S}$. By definition of the universal quantifier, the previous relation is $\neg\neg(\forall x)(\boldsymbol{S})$, from which follows $(\forall x)(x = x)$. It follows that, whatever x (even if x is a constant) we have $x = x$.

Theorem 2 says $(x = y) \iff (y = x)$. Let's show one implication. Assume $x = y$. We apply S6 to $y = x$, considered as a function of y. It says $x = x \iff y = x$, the conclusion follows by Theorem 1.

Theorem 3 says $(x = y \wedge y = z) \implies x = z$. The same argument as above says that if $x = y$, then $x = z \iff y = z$, making the theorem obvious.

The same argument shows that if $\boldsymbol{T} = \boldsymbol{U}$, and if $\boldsymbol{V}$ is a term (depending on a parameter z), then $\boldsymbol{V}\{\boldsymbol{T}\} = \boldsymbol{V}\{\boldsymbol{U}\}$. This is Criterion C44, and is known in Coq as *eq_ind*, while Scheme S6 is *eq_rec*. Theorem 1 is the definition *refl_equal*, other theorems are *sym_eq* and *trans_eq*.

There is no equivalent of Scheme S7 in Coq. The Leibnitz equality says that two objects x and y are equal iff every property which is true of x is also true of y. We shall later on define special terms called sets. They satisfy $x = y$ if $x \subset y$ and $y \subset x$. This makes equality weaker. In fact, if $x \neq y$, the middle excluded law implies that there exists some a such that either $a \in x$ and not $a \in y$ or $a \in y$ and not $a \in x$. Thus, assuming that 0 and 1 are sets, one of the following statements is true: there exists a such that $a \in 1$ and $a \notin 0$, or there exists b such that $b \in 0$ and $b \notin 1$, or $0 = 1$ (with our definition of integers as cardinals, the first assumption is true, but nothing can be said of a). In Bourbaki all terms are sets. In our work, we shall consider objects that are not sets. For instance, neither 1 nor 2 (considered as natural numbers) are sets. The relations $1 + 1 = 2$ and $1 \neq 2$ are the consequence of the fact that these objects have the same (or different) normal forms (modulo α-conversion).

Let $\mathrm{R}(z)$ be a relation. If $\mathrm{R}(x) = \mathrm{R}(y)$ implies $x = y$, then R is said *single-valued*. If moreover there exists x such that $\mathrm{R}(x)$ is true, then R is said *functional*. In this case R is equivalent to $x = \tau_x(\mathrm{R})$. Example. If $y > 0$ is an integer, there exists a unique x such that $y = x+1$. Denote by $p(y)$ the quantity $\tau_x(y = x + 1)$. We have then the conclusion: $y = x + 1$ if and only if $x = p(y)$. Thus $p(y_1) = p(y_2)$ if and only if $y_1 = y_2$, and we can forget the τ.

We shall add an axiom that says that two propositions are equal if they are equivalent. This is not possible in Bourbaki (since equality applies only to Sets). Let $\mathscr{T}$ and $\mathscr{T}'$ be two theories with and without this axiom. If T is a theorem of $\mathscr{T}$, consider T′, the same object where P = Q has been replaced by P $\iff$ Q. If we have a proof of T, and substitute = by $\iff$ whenever needed, we get of proof of T′ in $\mathscr{T}'$, because Coq uses only Theorems 1, 2 and 3, Scheme S6, Criterion C44, which are true for equivalence.

Notes. A reference of the form E.II.4.2 refers to [2], Theory of Sets, Chapter 2, section 4, subsection 2 (properties of union and intersection).

The document gives no proofs, except for the exercises. In order to show how difficult some theorems are, the numbers of lines of the proof is sometimes indicated in a comment.

Some statistics: there are 171 lemmas in jset, 98 in jfunc, 424 in set2 (correspondences), 364 in set3 and set31 (union; intersection, products) and 257 in set4 (equivalence relations).

In version 2, files jset and jfunc have been merged into set1, files set3 and set31 have also been merged. The number of theorems in these four files is now 279, 431, 375 and 257.

In Version 3, many trivial theorems have been removed, so that these numbers are respectively 202, 397, 338 and 242.

Chapter 2

Sets

This chapter describes the content of the file *set1.v*, that is an adaptation of the work of C. Simpson. It is formed of several modules, that will be commented one after the other. It implements the basis of the theory of sets; this is a logical theory (as described in the previous chapter) that contains a specific sign $\in$ and some rules about its usage; we must define the Coq equivalent *inc* and the associated rules.

2.1 Module Axioms

Types. In Coq, each object has a type, for instance 0 is of type *nat*, and the type of *nat* is *Set*; the type of a boolean like *True* is *Prop*; the type of *Set* or *Prop* is *Type*. Our sets will be of type *Set*[1]. For instance *nat* is a set, but neither 0, nor *True*, nor the abstraction $x \mapsto x+1$ nor a function is a set (functions are defined in the next chapter; more generally, no record containing sets is a set). The definition given here is not used anymore.

```
Definition Bset:=Set.
```

Is element of. The last axiom of Bourbaki states that there exists an infinite set. It is equivalent to the existence of the set of natural numbers and will be discussed in the second part of this report. The other axioms, as well as axiom scheme S8, use the symbols $\in$, $\subset$ or $\mathrm{Coll}_x \mathrm{R}$, that are not defined in Coq. The notation $x \subset y$ is a short-hand for:

$$(\forall z)((z \in x) \implies (z \in y)).$$

If $\boldsymbol{x}$ are $\boldsymbol{y}$ are two distinct letters, and $\boldsymbol{R}$ a relation that does not depend on $\boldsymbol{y}$, the relation

$$(\exists \boldsymbol{y})(\forall \boldsymbol{x})((\boldsymbol{x} \in \boldsymbol{y}) \iff \boldsymbol{R})$$

is denoted by $\mathrm{Coll}_{\boldsymbol{x}} \boldsymbol{R}$, and read as: the relation $\boldsymbol{R}$ is collectivizing in $\boldsymbol{x}$. The first axiom (axiom of extent) in Bourbaki says:

$$(\forall x)(\forall y)((x \subset y) \text{ and } (y \subset x)) \implies (x = y).$$

We can restate it as: if x and y are two sets, then $x = y$ if and only if $z \in x$ is equivalent to $z \in y$. As a consequence, if $\mathrm{R}(x)$ is collectivizing in x, there exists a unique set y such that $x \in y$ if and only if $\mathrm{R}(x)$ is true. It is denoted by $\{x, \mathrm{R}(x)\}$, or $\{x \,|\, \mathrm{R}(x)\}$ or $\mathscr{E}_x(\mathrm{R}(x))$.

[1] In the original version of C. Simpson, it was *Type*, so that *True* was a set; as a consequence there exists an element x that is in True but not in False, or vice-versa. This is not needed.

Some relations are not collectivizing, for instance $x \notin x$. In fact, if we assume that this is equivalent to $x \in y$, replacing x by y gives: $y \notin y$ is equivalent to $y \in y$, which is absurd. Almost all sets defined by Bourbaki are obtained by application of Axiom A3 (the relation "$x = a$ or $x = b$" is collectivizing), Axiom A4 (the relation $x \subset y$ is collectivizing) or Scheme S8 (Scheme of Selection and Union); a notable exception is the set of integers, for which a special axiom is required. Scheme S8 is a bit complicated. In [6], it is replaced by the axiom

$$(\forall x)(\exists y)(\forall z)(z \in y) \iff ((\exists t)(t \in x \text{ and } z \in t))$$

that asserts the existence of the union of sets, and the following scheme (Scheme of Substitution):

> If E is a relation that depends on x, y, $a_1, \ldots, a_k$, then for all x_1, x_2, ..., x_k, if we denote by $\mathrm{R}(x, y)$ the relation $\mathrm{E}(x, y, x_1, \ldots x_n)$, the assumption $(\forall x)(\forall y)(\forall y')\ \mathrm{R}(x, y) = \mathrm{R}(x, y') \implies y = y'$ implies that, for all t, the relation $(\exists u)(u \in t \text{ and } \mathrm{R}(u, v))$ is collectivizing in v.

The conclusion is the same as in S8. This scheme is more powerful than S8; for instance, it implies the axiom of the set of two elements A2. In fact we can deduce the existence of the empty set $\emptyset$ from this scheme (or from S8). Applying A4 to the empty set asserts the existence of a set that has a single element which is $\emptyset$, applying A4 again asserts the existence of a set t with two elements $\emptyset$ and $\{\emptyset\}$. If a and b are any elements, and $\mathrm{R}(u, v)$ is "$u = \emptyset$ and $v = a$ or $u = \{\emptyset\}$ and $v = b$", we get as conclusion: there exists a set formed solely of a and b. The assumption is clear: for fixed u, there is a unique v such that $\mathrm{R}(u, v)$. Question: can we apply S8 to this case? the answer is yes, provided that there exists a set X_a such that $a \in \mathrm{X}_a$ and a set X_b such that $b \in \mathrm{X}_b$. Such sets exist by virtue of Axiom A2. Hence A2 is required in Bourbaki, a conclusion of other axioms in [6]. The rules introduced below are closer to a Scheme of Substitution than to a Scheme of Selection and Union.

In the previous chapter, we have given a proof with seven lines that says $1 + 1 = 2$. The analogue proof is trivial in Coq (both objects have the same normal form *SSO*). We have also seen that the induction principle for integers in Bourbaki is the same as that of integers in Coq; as a consequence, if we can identify the Coq integers with the integers of Bourbaki, then a lot of theorems will become trivial (i.e., are already proved by someone else). For this reason, all types, such as *nat*, will be a set. The Coq notation *O:nat* says that O is of type nat, we want it to be the same as $z \in \mathbb{N}$ where z stands for 0 and $\mathbb{N}$ for *nat*. In Bourbaki, $z \in \mathbb{N}$, $\mathbb{N} \in z$ and $\mathbb{N} \in \mathbb{N}$ are legal statements; the first one is true, the other ones are false. In order to avoid paradoxes, people sometimes use a hierarchy of sets: if $x \in y$ and x is at level n, then y must be at level $n + 1$. Thus $x \in x$ is illegal or false. The axiom of foundation states that if x is not empty, there exists $y \in x$ such that the intersection of x and y is empty; it forbids the existence of a sequence of sets with $x_i \in x_{i+1}$ and $x_n \in x_0$. This axiom is independent of all other ones. We shall not use it. However, in Coq there is a type hierarchy, so that there is no a whose type is itself, and no pairs a and b such that a is of type b and b of type a. In [6] there is a theorem that says: if the axiom of foundation is true, then X is an ordinal if and only if for all u and v in X we have $u \in v$ or $u = v$ or $v \in u$ and for all $u \in \mathrm{X}$ we have $u \subset \mathrm{X}$. Such a statement becomes impossible in the case of a set hierarchy.

We have the property that *O:nat* is the same as $z \in \mathbb{N}$ if we chose $\mathbb{N}$ to be *nat* and z a function of 0, say $\mathscr{R}0$. We postulate the existence of such a function; note that the type of 0 is *nat* while the type of $\mathscr{R}0$ is *Set*, i.e. the same as (in fact, compatible with) the type of *nat*. Let's define $\mathbb{N}_2$ as the set of even numbers; we have $\mathbb{N}_2 \subset \mathbb{N}$, and $z \in \mathbb{N}_2$. Let 0_2 be zero in the type *even*. We have $z = \mathscr{R}0 = \mathscr{R}0_2$. This relation will be a consequence of the definition of 0_2. We

want $\mathscr{R}0 \neq \mathscr{R}1$, since otherwise this mechanism is useless. Thus we introduce a parameter and an axiom. Later on[2] we shall define $\mathscr{R}0$ and $\mathscr{R}1$; we shall prove that the axiom is satisfied in this case.

```
Parameter Ro : forall x : Set, x -> Set.
Axiom R_inj : forall (x : Set) (a b : x), Ro a = Ro b -> a = b.
```

We define '$x \in y$' to be: there is an object a of type y such that $\mathscr{R}a = x$. Inclusion $x \subset y$ is defined as in Bourbaki. These two operations are called *In* or *Included* in *Ensembles*, but *inc* or *sub* in our framework.

```
Definition inc (x y : Set) := exists a : y, Ro a = x.
Definition sub (a b : Set) := forall x : E, inc x a -> inc x b.
```

Extensionality. The axiom of extent is the same as in Bourbaki: if $x \subset y$ and $y \subset x$ then $x = y$. We add another one for functions; if two functions take the same values everywhere we declare them equal.

```
Axiom extensionality : forall a b : Set, sub a b -> sub b a -> a = b.

Axiom prod_extensionality :
  forall (x : Type) (y : x -> Type) (u v : forall a : x, y a),
    (forall a : x, u a = v a) -> u = v.

Lemma arrow_extensionality :
  forall (x y : Type) (u v : x -> y), (forall a : x, u a = v a) -> u = v.
```

Given a type t or a set x, it will be declared *nonempty* if there is a witness, i.e., an instance of t or an element of x.

```
Inductive nonemptyT (t : Type) : Prop :=
    nonemptyT_intro : t -> nonemptyT t.
Inductive nonempty (x : Set) : Prop :=
    nonempty_intro : forall y : Set, inc y x -> nonempty x.
```

The axiom of choice. We assert the existence of a function $\mathscr{C}_\mathrm{T}$ of two parameters p and q, depending on a type t, that satisfies the following property: if there exists an object x of type t such that $p(x)$ is true, then $c = \mathscr{C}_\mathrm{T}(p,q)$ is an object of type t such that $p(c)$ is true. Note. If we take for p the constant function True, we get: if there exists an element of type t, then c_q is of type t. But, by construction, c_q is of type t. From this, one could deduce that every type is non-empty. Our definition requires that q be of type t, so that we get: if q is of type t, then c_q is of type t; which sounds better. We emphasize that our choice function depends both on p and q. This is the equivalent of Bourbaki's τ.

```
Parameter chooseT : forall (t : Type) (p : t -> Prop)(q:nonemptyT t),  t.

Axiom chooseT_pr :
  forall (t : Type) (p : t -> Prop) (ne : nonemptyT t),
    ex p -> p (chooseT p ne).
```

[2] Only in version 1 of the software

Images. The scheme of selection and union is the following: Given a relation R(x, y); assume that for fixed y, we have a set E_y such that R(x, y) implies $x \in E_y$. Then, for every Y, there is a set Z_Y containing all x for which there is an $y \in Y$ such that R(x, y). A simple case is when R is independent of y. Another simple case is when R has the form $x = f(y)$. The axiom of the set of two elements (shown later) says that we can select $E_y = \{f(y)\}$. As a consequence the image of a set by a function is a set. We define here a parameter *IM*, and the corresponding axioms.

```
Parameter IM : forall x : Set, (x -> Set) -> Set.

Axiom IM_exists :
          forall (x : Set) (f : x -> Set) (y : Set),
          inc y (IM f) -> exists a : x, f a = y.
Axiom IM_inc :
          forall (x : Set) (f : x -> Set) (y : Set),
          (exists a : x, f a = y) -> inc y (IM f).
```

Double negation axiom. A property is either true or false. Thus two sets are equal or not.

```
Axiom excluded_middle : forall P : Prop, ~ ~ P -> P.

Lemma excluded_middle': forall A B:Prop, (~ B->A)->(~A->B).
Lemma p_or_not_p : forall P : Prop, P \/ ~ P.
Lemma equal_or_not: forall  x y:Set, x= y \/ x<> y.
Lemma inc_or_not: forall  x y:Set, inc x y \/ ~ (inc x y).
```

This axiom was introduced by C. Simpson, and has been withdrawn in Version 3. We shall explain below why it was used.

```
(* Axiom proof_irrelevance : forall (P : Prop) (q p : P), p = q. *)
```

We already explained that P = Q is the same as P $\iff$ Q for propositions; the idea is that we can use the *rewrite* and *autorewrite* tactics.

```
Axiom iff_eq : forall P Q : Prop, (P -> Q) -> (Q -> P) -> P = Q.
```

Special realizations. The four axioms of this paragraph have been withdrawn in the second edition, since they are not needed for our purpose. Moreover, in Vezrsion 3, we changed the type of a set, so that there are currently ill-typed.

The next two axioms define inductively $\mathscr{R}i$ for natural numbers i. In the case of $i = 0$, the value is $\emptyset$, in the case of $i+1$ it is $\mathscr{R}i \cup \{\mathscr{R}i\}$; note that the emptyset, singleton, and union will be defined later, so that $\mathscr{R}i$ is defined by extensionality.

```
Axiom nat_realization_0 : forall x : Set, ~ inc x (Ro 0).
Axiom nat_realization_S :
          forall (n : nat) (x : Set),
          inc x (Ro (S n)) = (inc x (Ro n) \/ x = Ro n).
```

These axioms say that $\mathscr{R}$P is P for every proposition, and $\mathscr{R}p$ is the empty set whenever p is a proof of True. We shall see later one what this implies.

```
Axiom prop_realization : forall x : Prop, Ro x = x.
Axiom true_proof_realization_empty : forall t : True, Ro t = Ro 0.
```

2.2 Module constructions

We denote by EE, EEE, EP, EEP, the type of functions with one or two arguments, that return a set or a boolean. The definition that follows says that p is a predicate, satisfied by a unique object of type t. This will be extended later to objects of different types, for instance functions[3].

```
Definition exists_unique t (p : t->Prop) :=
  (ex p) & (forall x y : t, p x -> p y -> x = y).
```

This defines $x \subsetneq y$.

```
Definition strict_sub (a b : Set) :=  (neq a b) & (sub a b).
```

These lemmas say that $x \subset x$, and if $x \subset y$ and $y \subset z$, then $x \subset z$; if one $\subset$ is replaced by $\subsetneq$ in the assumption, then the same holds in the conclusion.

```
Lemma sub_refl : forall x, sub x x.
Lemma sub_trans : forall a b c, sub a b -> sub b c -> sub a c.
Lemma strict_sub_trans1 :
  forall a b c, strict_sub a b -> sub b c -> strict_sub a c.
Lemma strict_sub_trans2 :
  forall a b c, sub a b -> strict_sub b c -> strict_sub a c.
```

Empty sets and types. We say that a set is *empty* if it has no element; by extensionality, it is unique. Bourbaki proves existence of the empty set by noting that for every set y, and every property P, there is a set containing all elements of y with the property P. In particular, we can define the complement of x in y; taking $x = y$ gives the empty set. In Coq, the situation is simpler: we define $\emptyset$ as a type without constructor, hence there is no $a \in x$, since there is no $b : x$. In version 3, the empty set is also *Empty_set*.

```
Definition empty (x : Set) := forall y : Set, ~ inc y x.
Inductive emptyset : Set :=.
```

Given a set x, we have $b \in x$ if $b = \mathscr{R}a$ for some $a : x$. As a consequence, if $a : x$ then $\mathscr{R}a \in x$. In particular, x is not empty. On the other hand, if $b \in x$ there is an a with $a : x$.

```
Lemma R_inc : forall (x : Set) (a : x), inc (Ro a) x.
Lemma nonemptyT_not_empty : forall x : E, nonemptyT x -> ~ empty x.
Lemma inc_nonempty : forall x y, inc x y -> nonemptyT y.
Lemma sub_emptyset_any:  forall x, sub emptyset x.
```

An inverse for $\mathscr{R}$. We define a function $\mathscr{B}$ that takes 3 arguments, x, y and H, two sets and a proof of $x \in y$. The first two arguments are implicit: they are deduced from the type of H. We shall sometimes write $\mathscr{B}(\mathrm{H} : x \in y)$. The function uses the axiom of choice $\mathscr{C}_{\mathrm{T}}(p,q)$ to select an object a of type y such that $p(a)$, namely $\mathscr{R}a = x$. Assumption H says that such an object exists, and as a consequence it implies q, a proof that the type y is inhabited. Thus $p(\mathscr{B})$ is true, i.e.,

$$\mathscr{R}(\mathscr{B}(\mathrm{H} : x \in y)) = x.$$

[3] In version 1, we defined *neq* and *elt* corresponding to $x \neq y$ and $y \in x$.

If we replace x by $\mathscr{R}z$, we get $\mathscr{R}(\mathscr{B}(\mathrm{H})) = \mathscr{R}z$, hence, by injectivity

$$\mathscr{B}(\mathrm{H} : \mathscr{R}z \in y) = z.$$

```
Definition Bo (x y : Set) (hyp : inc x y) :=
  chooseT (fun a : y => Ro a = x) (inc_nonempty hyp).

Lemma B_eq : forall x y (hyp : inc x y), Ro (Bo hyp) = x.
Lemma B_back : forall (x:Set) (y:x) (hyp : inc (Ro y) x), Bo hyp = y.
```

If H is a proof of $y \in x$, then $\mathscr{B}$ H is of type x; in particular the type x cannot be empty, and x cannot be the empty set. The proof of the next lemma uses the excluded-middle axiom in order to replace $\neg\forall y \neg \mathrm{P}$ by $\exists y$ P, where P is $y \in x$. Finally, we have a lemma that says that a set is either empty or nonempty.

```
Lemma not_empty_nonemptyT : forall x, ~ empty x -> nonemptyT x.
Lemma emptyset_pr: forall x, ~ inc x emptyset .
Lemma is_emptyset: forall x, (forall y, ~ (inc y x)) -> x = emptyset.
Lemma emptyset_dichot : forall x, (x = emptyset \/ nonempty x).
```

Reasoning by cases. Let T be a type, P a proposition, a a function that maps a proof of P into T and b a function that maps a proof of $\neg$P into T. Since P is true or false, there is a proof p of P or a proof q of its negation, thus $a(p)$ or $b(q)$ is an object of type T; as a consequence the type T is non-empty. Let HT be the assumption that $a(p)$ is independent of the proof p of P and HF the assumption that $b(q)$ is independent of the proof of $\neg$P. We use the axiom of choice to construct $\mathscr{C}_{\mathrm{C}}(a,b)$, an object of type T, such that for every proof p of P we have $a(p) = x$ and for every proof q of $\neg$P we have $b(q) = x$.

```
Definition by_cases_pr (T : Type) (P : Prop) (a : P -> T)
  (b : ~ P -> T) (x : T) :=
  (forall p : P, a p = x) & (forall q : ~ P, b q = x).

Lemma by_cases_nonempty :
 forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T), nonemptyT T.
Definition by_cases (T : Type) (P : Prop) (a : P -> T)  (b : ~ P -> T) :=
  chooseT (fun x : T => by_cases_pr a b x) (by_cases_nonempty a b).
```

Assume that p is a proof of P, and HT is is true. Then HF is true, since there is no proof of $\neg$P. Thus $\mathscr{C}_{\mathrm{C}}(a,b) = p(a)$. In the same fashion, if q is a proof of $\neg$P and HF is true, then $\mathscr{C}_{\mathrm{C}}(a,b) = q(b)$.

In version 1 (see section 9.3), we used the proof-irrelevance axiom, that says essentially that HF and HT are true, for all propositions P. Removing this axiom required two changes: In the case of *Xo* below, we use injectivity of $\mathscr{R}$. In the file set3, we define the intersection of a family of sets as the set $\{y \in \mathrm{X}_i, \forall\, j \in \mathrm{I}, y \in \mathrm{X}_j\}$; it depends on a proof that I is non-empty (and this gives us an index $i \in \mathrm{I}$). We must show that this set is independent of i, which is obvious.

```
Lemma by_cases_if :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (p : P),
    (forall p1 p2: P, a p1 = a p2) ->
    by_cases a b = a p.
Lemma by_cases_if_not :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (q : ~ P),
    (forall p1 p2: ~P, b p1 = b p2) ->
    by_cases a b = b q.
```

Choosing a representative or the empty set. We simplified a bit the original code (see section 9.3). We construct $\mathscr{C}(p)$, that behaves like $\tau_x(\mathrm{P})$: if there is an x with $p(x)$, then $p(\mathscr{C}(p))$ is true; we add the condition that $\mathscr{C}(p)=\emptyset$, if there is no such x.

```
Definition choose (p:EP) :=
  chooseT (fun x => (ex p -> p x) & ~(ex p) -> x = emptyset)
  (nonemptyT_intro emptyset).

Lemma choose_pr : forall p, ex p -> p (choose p).
Lemma choose_not : forall p, ~(ex p) -> choose p = emptyset.
```

We consider a variant, $\mathscr{C}_{\mathbb{N}}(p)$, where $x\in\mathbb{N}$; if no element satisfies p then $\mathscr{C}_{\mathbb{N}}(p)=0$.

```
Definition choosenat p :=
  chooseT (fun u => (ex p -> p u)  &  ~(ex p) -> u = 0)  (nonemptyT_intro 0).
Lemma choosenat_pr : forall p, ex p -> p (choosenat p).
```

Representatives of nonempty sets. Fix z. Let $p(x)$ be the property $x\in z$. If z is not empty, there is an y such that $p(y)$, hence $\mathscr{C}(p)$ satisfies p, this is an element of z. This will be denoted by *rep z*.

```
Definition rep (x : Set) := choose (fun y : Set => inc y x).

Lemma nonempty_rep : forall x, nonempty x -> inc (rep x) x.
```

The first of these lemmas is obvious; the second has already been used; it relies on the excluded-middle axiom.

```
Lemma not_exists_pr : forall p : EP, (~ ex p) = (forall x : Set, ~ p x).
Lemma exists_proof : forall p : EP, ~ (forall x : Set, ~ p x) -> ex p.
```

Defining a term depending on a boolean value. This defines a function $\mathscr{Y}(\mathrm{P},x,y)$ via $\mathscr{C}_{\mathrm{C}}(f,g)$. Here P is a property, f the function that returns x for every proof of P and g the function that returns y for every proof of $\neg$P. As a consequence $\mathscr{Y}(\mathrm{P},x,y)$ is x if P is true and y otherwise.

```
Definition Yo : Prop -> EEE :=
  fun P x y => by_cases (fun _ : P => x) (fun _ : ~ P => y).

Lemma Y_if : forall (P : Prop) (hyp : P) x y z, x = z -> Yo P x y = z.
Lemma Y_if_not : forall (P : Prop) (hyp : ~ P) x y z, y = z -> Yo P x z = y.

Lemma Y_if_rw : forall (P:Prop) (hyp :P) x y, Yo P x y = x.
Lemma Y_if_not_rw : forall (P:Prop) (hyp : ~P) x y, Yo P x y = y.
```

A variant[4] where the arguments are of type A instead of being a set.

```
Definition Yt (A:Type): Prop -> (A->A->A):=
  fun P x y => by_cases (fun _ : P => x) (fun _ : ~ P => y).

Lemma Yt_if_rw : forall (A:Type)(P : Prop) (hyp : P) (x:A) y,
  Yt P x y = x.
Lemma Yt_if_not_rw :  forall (A:Type) (P : Prop) (hyp : ~ P) x  (y:A),
  Yt P x y = y.
```

[4] New in version 3

Set of elements such that P. In Bourbaki, the "Scheme of selection and union" is the following : we have four distinct variables x, y, X and Y, and a relation R that depends on x and y, but not on X, Y. The assumption is $\forall y, \exists X, \forall x, R \implies x \in X$. The conclusion is that for every Y, the relation $\exists y, y \in Y \wedge R$ is collectivizing in x. Said otherwise, for every Y, there is a set Z such that $x \in Z$ is equivalent to the existence of $y \in Y$ such that R. A simple case is when R does not depend on y. Then, the assumption $\forall x, R(x) \implies x \in X$ implies the existence of Z such that $x \in Z$ is equivalent to $R(x)$. In particular, if $Q(x)$ is any relation, there is a set Z such that $x \in Z$ is equivalent to $x \in Y \wedge Q(x)$. Here is the Coq implementation.[5]

```
Record Zorec (x : Set) (f : x -> Prop) : Set :=
    {Zohead : x; Zotail : f Zohead}.
Definition Zo  := fun (x:Set) (p:EP)
  => let P := Zorec (fun a : x => p (Ro a)) in IM (fun t : P => Ro (Zohead t)).
```

In version 3, we changed a bit the definition; it is now:

```
Inductive Zorec (x : Set) (f : x -> Prop) : Set :=
   Zorec_c : forall a: x, f a -> Zorec f.
Definition Zo (x:Set) (p:EP) :=
  let f:= fun a : x => p (Ro a) in
    IM (fun  (z : Zorec f) => let (a, _) := z in Ro a).
```

The object $\mathscr{Z}(x,p)$ is the image of some function g. This means that $y \in \mathscr{Z}(x,p)$ if and only if there is a t such that $y = g(t)$. If t is the record defined by a, it holds a of type x and $p(\mathscr{R}a)$. The function $g(t)$ is then $\mathscr{R}a$. Thus $y = g(t)$ says $y \in x$; it implies $p(y)$. The reverse is true.

The set is denoted in Bourbaki by $\mathscr{E}_x$(P and $x \in$ A). In the French version, it is denoted by $\{x \mid$ P and $x \in$ A$\}$; Bourbaki notes that this may be abbreviated as $\{x \in$ A $\mid$ P$\}$.

```
Lemma Z_inc :  forall x p y, inc y x -> p y -> inc y (Zo x p).
Lemma Z_sub :  forall x p,   sub (Zo x p) x.
(* Lemma Z_pr :   forall x p y, inc y (Zo x p) -> p y. *)
Lemma Z_all :  forall x p y, inc y (Zo x p) ->  (inc y x) & (p y).
```

This defines an auxiliary function *Yy*, with arguments f and x. It depends on a property P. If p is a proof of P, the function returns $f(p)$, and if p is false, it returns x.[6]

```
Definition Yy : forall P : Prop, (P -> Set) -> EE :=
  fun P f x => by_cases f (fun _ : ~ P => x).

Lemma Yy_if :
 forall (P : Prop) (hyp : P) (f : P -> Set) x z,
   (forall p1 p2: P, f p1 = f p2) ->
   f hyp = z -> Yy f x = z.
Lemma Yy_if_not :
  forall (P : Prop) (hyp : ~ P) (f : P -> Set) x z, x = z -> Yy f x = z.
```

We define here $\mathscr{X}(f,y)$ as $Yy(g,\emptyset)$. Given a proof H of $y \in x$, $\mathscr{B}$H is of type x; we assume that f is defined for such objects, and pose g(H) $= f(\mathscr{B}$H$)$. This is $\mathscr{X}(f,y)$ by definition. If $y = \mathscr{R}z$, where z is of type x, we know $\mathscr{B}$H $= z$. Said otherwise: $\mathscr{X}(f)$ is some function F, defined for every y, whose value is $\emptyset$ if $y \notin x$, and F$(\mathscr{R}z) = f(z)$ if $z : x$.

[5] In the orginal version, the type of f was $x \to \mathscr{E}$. This is no more possible, since a proposition is not a set.

[6] In version 3, we added the requirement that $f(p)$ be independent of p.

```
Definition Xo (x : Set) (f : x -> Set) (y : Set) :=
  Yy (fun hyp : inc y x => f (Bo hyp)) emptyset.

Lemma X_eq :
  forall (x : Set) (f : x -> Set) (y z : Set),
    (exists a : x,  (Ro a = y) & (f a = z)) -> Xo f y = z.

Lemma X_rewrite : forall (x : Set) (f : x -> Set) (a : x), Xo f (Ro a) = f a.
```

Cuts. Let p be the property that x is even. The construction *cut* allows us to define $\mathbb{N}_2$ as the set of all even integers. This means $y \in \mathbb{N}_2$ if $\mathscr{R}y$ is even. Since 0 is even we have $z \in \mathrm{Z}_2$. Let 0_2 be the object of type $\mathbb{N}_2$ such that $\mathscr{R}0_2 = z$. The construction *cut_to* take 0_2 as argument and returns 0. Later on, we shall see that for any inclusion like $\mathbb{N}_2 \subset \mathbb{N}$ there is a function of type $\mathbb{N}_2 \to \mathbb{N}$.

We define *cut x p* as the set of all elements $y \in x$ such that if H is a proof of $y \in x$, and $z = \mathscr{B}\mathrm{H}$, then $p(z)$ is true, (here p is a property on the type x). Note that $\mathscr{R}z = y$. Argument x is implicit.

```
Definition cut (x : Set) (p : x -> Prop) :=
  Zo x (fun y : Set => forall hyp : inc y x, p (Bo hyp)).

Lemma cut_sub : forall (x : Set)(p : x -> Prop), sub (cut p) x.
Lemma cut_inc : forall (x : Set)(p : x -> Prop)(y : x), p y -> inc (Ro y) (cut p).
```

Let y be of type *cut p*. In this case *R_inc y* is a proof that $\mathscr{R}y \in \mathrm{C}$. If we apply *cut_sub* we get a proof that $\mathscr{R}y \in x$. We apply $\mathscr{B}$ to this. If this gives z, then z is of type x, $\mathscr{R}z = \mathscr{R}y$ and $p(z)$ is true.

```
Definition cut_to (x : Set) (p : x -> Prop) (y : cut p) :=
  Bo (cut_sub (p:=p) (R_inc y)).

Lemma cut_to_R_eq : forall (x:Set)(p: x -> Prop) (y: cut p), Ro (cut_to y) = Ro y.
Lemma cut_pr : forall (x : Set) (p : x -> Prop) (y : cut p), p (cut_to y).
```

Let f be a function defined on the type a. If x is of type *IM f*, there is some $y : a$ such that $f(y) = \mathscr{R}x$. The function *IM_lift* uses the axiom of choice, and returns such an y.

```
Definition IM_lift : forall (a : Set) (f : a -> Set), IM f -> a.
 ir. assert (exists x : a, f x = Ro H). ap IM_exists. ap R_inc.
 assert (nonemptyT a). induction H0. app nonemptyT_intro.
 exact (chooseT (fun x : a => f x = Ro H) H1).
Defined.

Lemma IM_lift_pr :
  forall (a : Set) (f : a -> Set) (x : IM f), f (IM_lift x) = Ro x.
```

2.3 Module Little

We define a *singleton* as the image of a set with one element. We could proceed as in Bourbaki, namely to define it as a *doubleton x x*. We denote this as $\{x\}$. By construction $z \in \{x\} \iff z = x$. From this one can deduce that a singleton is nonempty, and we have an extensionality property.

```
Inductive one_point : Set :=
    one_point_intro : one_point.

Definition singleton (x : Set) := IM (fun p : one_point => x).

Lemma singleton_inc : forall x, inc x (singleton x).
Lemma singleton_eq : forall x y, inc y (singleton x) -> y = x.
Lemma singleton_inj : forall x y, singleton x = singleton y -> x = y.
Lemma singleton_rw: forall x y, inc y (singleton x) = (y = x).
Lemma nonempty_singleton: forall x, nonempty (singleton x).
Lemma sub_singleton: forall x y, inc x y -> sub (singleton x) y.
```

It is trivial to construct an object *two_points* with two constructors. This gives a set with two distinct elements. We call this the canonical doubleton, the elements will be *TPa* and *TPb*.

```
Inductive two_points : Set :=  | two_points_a | two_points_b.

Definition TPa := Ro two_points_a.
Definition TPb := Ro two_points_b.

Lemma two_points_pr: forall x,
  inc x two_points = (x= TPa \/ x = TPb).
Lemma two_points_distinct: TPa <> TPb.
Lemma two_points_distinctb: TPb <> TPa.
```

Given two elements x and y, we construct a set, a *doubleton*, denoted by $\{x, y\}$, satisfying $z \in \{x, y\} \iff z = x \vee z = y$, as the image of the canonical doubleton. In Bourbaki, Axiom A2 says that such a set exists. By extensionality, *two_points* is a doubleton.

```
Definition doubleton (x y : Set) :=
  IM (fun t => two_points_rect (fun _ : two_points => Set) x y t).

Lemma doubleton_first : forall x y, inc x (doubleton x y).
Lemma doubleton_second : forall x y, inc y (doubleton x y).
Lemma doubleton_or : forall x y z, inc z (doubleton x y) -> z = x \/ z = y.
Lemma doubleton_rw : forall x y z : Set,
  inc z (doubleton x y) = (z = x \/ z = y).

Lemma two_points_pr2: doubleton TPa TPb =  two_points.

Lemma doubleton_inj :  forall x y z w : Set,
    doubleton x y = doubleton z w ->  (x = z & y = w) \/  (x = w & y = z).

Lemma doubleton_singleton : forall x, doubleton x x = singleton x.
```

If $x \in z$ and $y \in z$ then $\{x, y\} \subset z$. A doubleton is nonempty. We have $\{x, y\} = \{y, x\}$.

```
Lemma nonempty_doubleton: forall x y, nonempty (doubleton x y).
Lemma sub_doubleton: forall x y z,
  inc x z -> inc y z -> sub (doubleton x y) z.
Lemma doubleton_symm: forall a b,
  doubleton a b = doubleton b a.
```

2.4 Module Basic Realization

This module is not used for the Bourbaki Project. It has been withdrawn in version 2 as well as the axioms associated to it.

The first lemma says $\mathscr{R}0 = \emptyset$; as noted above, we could use this as the definition of $\mathscr{R}0$. The second lemma says that $\emptyset$ is *False*. Note that the objects have different types, but are equal by extensionality. The realization of *False* is itself, hence the empty set. The realization of every t of type *True* is the empty set. By extensionality, this implies that *True* is the singleton $\{\emptyset\}$. The realization of *True* is itself.

```
Lemma nat_zero_emptyset : Ro 0 = emptyset.
Lemma false_emptyset : emptyset = False.
Lemma R_false_emptyset : Ro False = emptyset.
Lemma true_proof_emptyset : forall t : True, Ro t = emptyset.
Lemma true_singleton_emptyset : singleton emptyset = True.
Lemma R_true_singleton_emptyset : Ro True = singleton emptyset.
```

By definition $\mathscr{R}1 = \emptyset \cup \{\emptyset\} = \{\emptyset\}$, since $\mathscr{R}0 = \emptyset$. Moreover $\mathscr{R}2$ has two elements, which are $\emptyset$ and $\{\emptyset\}$ said otherwise, True and False. Since every proposition is True or False, we get that $\mathscr{R}2$ is Prop.

```
Lemma R_one_singleton_emptyset : Ro 1 = singleton emptyset.
Lemma R_two_prop : Ro 2 = Prop.
```

2.5 Module Complement

The *complement* in a of b, denoted $a \setminus b$ or sometimes $a - b$, is the subset of a formed of elements not in b; it is the set of all elements in a but not in b. If x is in a but not in $a \setminus b$, then it is in b. If $a \setminus b$ is empty, then $a \subset b$.

```
Definition complement (a b : Set) := Zo a (fun x : Set => ~ inc x b).

Lemma inc_complement :
 forall a b x, inc x (complement a b) = (inc x a & ~ inc x b).
Lemma use_complement :
 forall a b x, inc x a -> ~ inc x (complement a b) -> inc x b.
```

These lemmas are obvious. If A $\subset$ E then E $-$ (E $-$ A) = A. We have E $-$ E = $\emptyset$ and E $-\emptyset$ = E.

```
Lemma sub_complement: forall a b, sub (complement a b) a.
Lemma strict_sub_nonempty_complement : forall x y,
  strict_sub x y -> nonempty (complement y x).

Lemma double_complement: forall a x,
  sub a x -> complement x (complement x a) = a.
Lemma complement_itself : forall x,  complement x x = emptyset.
Lemma complement_emptyset : forall x,  complement x emptyset = x.
Lemma empty_complement: forall a b,
  complement a b = emptyset -> sub a b.
Lemma not_inc_complement_singleton: forall a b,
  ~ (inc b (complement a (singleton b))).
```

2.6 Module Pair

We define here an operator *bpair*[7], and the associated two projectors *pr1* and *pr2*. Given two sets x and y, the pair z associated to x and y is denoted by (x,y) and the projectors satisfy $\mathrm{pr}_1 z = x$ and $\mathrm{pr}_2 z = y$. Note: in Coq, the expression *(x,y)* is a *pair*; this is however not a set. The pair constructor will be denoted by J and the projectors by P and Q.

In the English version [2], Bourbaki uses Axiom A3 to show the existence of such an object. In the French version [3], he uses the doubleton $\{x,\{x,y\}\}$. In the definition here, we use the doubleton $\{\{x\},\{\emptyset,\{y\}\}\}$. This definition is a bit complicated, but it has a strange property: a pair is a set with two distinct elements.

```
Definition pair_first (x y:Set):= singleton x.
Definition pair_second (x y:Set):= doubleton emptyset (singleton y).

Definition bpair (x y : Set) :=
  doubleton (pair_first x y) (pair_second x y).
Notation J := bpair.

Definition is_pair (u : Set) := exists x, exists y, u = J x y.

Lemma pair_distincta:forall x y z w,
  (pair_first x y = pair_second z w)-> False.
Lemma pair_distinct:forall x y,
  pair_second x y <>  pair_first x y.
```

Whatever definition is chosen, a pair must satisfy the two following properties:

```
Lemma pr1_injective: forall a b c d, J a b = J c d -> a = c.
Lemma pr2_injective: forall a b c d, J a b = J c d -> b = d.
```

The previous lemmas allows us to defined the destructors via the axiom of choice.

```
Definition pr1 (u : Set) :=
  choose (fun x : Set => ex (fun y : Set => u = J x y)).
Definition pr2 (u : Set) :=
  choose (fun y : Set => ex (fun x : Set => u = J x y)).

Notation P := pr1.
Notation Q := pr2.
```

The next lemmas say that if z is a pair, then z is the pair $(\mathrm{pr}_1 z, \mathrm{pr}_2 z)$, the quantity (x,y) is a pair, $\mathrm{pr}_1(x,y) = x$, $\mathrm{pr}_2(x,y) = y$. Moreover, if $(x,y) = (x',y')$ then $x = x'$ and $y = y'$, and conversely two pairs with the same projectors are equal.[8]

```
Lemma pair_recov : forall u, is_pair u -> J (P u) (Q u) = u.
Lemma pair_is_pair : forall x y, is_pair (J x y).
Lemma is_pair_rw : forall x, is_pair x = (x = J (P x) (Q x)).
Lemma pr1_pair : forall x y, P (J x y) = x.
Lemma pr2_pair : forall x y, Q (J x y) = y.

Lemma pair_extensionality : forall a b,
  is_pair a -> is_pair b -> P a = P b -> Q a = Q b -> a = b.
```

[7] Was *pair* in the first version

[8] Definition of *pair_recovers* was removed in V3; it was *J (P u) (Q u) = u.*

A set of pairs is sometimes called a graph (or a relation in the original work of C. Simpson). We denote by $\mathcal{V}(x,g)$ an element y, if it exists, such (x,y) is in the graph g. If there is an y such that (x,y) is in the graph, then $(x,\mathcal{V}(x,g))$ is in the graph. Otherwise, $\mathcal{V}(x,g)=\emptyset$. Later on, we shall define the domain as the set all x for which there a y; a graph is said functional (on its domain E) if for every x (in the set E) there is a unique y such that (x,y) is in the graph.

```
Definition V (x f : Set) := choose (fun y : Set => inc (J x y) f).

Lemma V_inc : forall x z f,
 (exists y, inc (J x y) f) -> z = V x f -> inc (J x z) f.

Lemma V_or : forall x f,
  (inc (J x (V x f)) f) \/
  ((forall z, ~(inc (J x z) f)) & (V x f = emptyset)).
```

2.7 Module Image

This module contained initially 8 lemmas, and only 2 of them will be used in what follows. If f is a mapping, x a set, we denote the image of x by f as $f\langle x\rangle$.

```
Definition fun_image (x : Set) (f : EE) := IM (fun a : x => f (Ro a)).

Lemma inc_fun_image : forall x f a, inc a x -> inc (f a) (fun_image x f).
Lemma fun_image_rw : forall  f x y,
  inc y (fun_image x f) = exists z, (inc z x & f z = y).
```

2.8 Module Powerset

Bourbaki introduces an axiom that says that for every set x, there is a set y, the *powerset* of x, denoted $\mathfrak{P}(x)$ containing the subsets of x. C. Simpson considers the set of cuts[9]. Remember that the *cut* of a predicate p is the set of all y such that $p(x)$ is true. For each subset y of x there is a predicate whose cut is y. In Version 3, we changed the type of a set. As a consequence, Coq refuses the definition. Instead of a predicate, we use a function with values A and B, and consider the set of all t such that $p(t)=\mathrm{A}$.

```
(* Definition powerset (x : Set) := IM (fun p : x -> Prop => cut p). *)
Definition powerset (x : Set) :=
  IM (fun p : x -> two_points =>
    Zo x (fun y : Set => forall hyp : inc y x, p (Bo hyp) = two_points_a)).
```

The characteristic property is that $y\subset x$ if and only if $y\in\mathfrak{P}(x)$.

```
Lemma powerset_inc : forall x y, sub x y -> inc x (powerset y).
Lemma powerset_sub : forall x y, inc x (powerset y) -> sub x y.
Lemma powerset_inc_rw : forall x y, inc x (powerset y) = sub x y.
Lemma inc_x_powerset_x: forall x, inc x (powerset x).
```

[9] Definition changed in V3

2.9 Module Union

Bourbaki defines the *union* $\bigcup_{\iota\in I} X_\iota$ of a family of sets. This means that we have a set I and a mapping $\iota \mapsto X_\iota$ defined for $i \in I$. If the mapping is the identity, which is the case considered here, we get the union of a set of sets, denoted by $\bigcup X$. The union exists as a direct consequence of S8 (Scheme of Selection and Union). The assumption is $(\forall i)(\exists Z)(\forall x)(i \in I \text{ and } x \in X_i) \implies x \in Z$ (take $Z = X_i$). The conclusion is the existence of a set containing all elements satisfying $(\exists i)(i \in I \text{ and } x \in X_i)$. Instead of an axiom, we use the following construction:

```
Record Union_integral (x : Set) : Set :=
  {Union_param : x; Union_elt : Ro Union_param}.
Definition union (x : Set) :=
  IM (fun i : Union_integral x => Ro (Union_elt (x:=x) i)).
```

A *Union_integral* record contains two fields, say p and e. Let $q = \mathscr{R}p$. Since p is type x, we have $q \in x$. An object y is in the union if $y = \mathscr{R}e$ for some integral record. Since e is of type q, this means $y \in q$. The next two lemmas are then obvious; the third one is easy. Bourbaki considers the union of subsets of x; according to the last lemma, this is a subset of x.

```
Lemma union_inc : forall x y a, inc x y -> inc y a -> inc x (union a).
Lemma union_exists : forall x a,
    inc x (union a) -> exists y,  inc x y & inc y a.

Lemma union_sub: forall x y, inc x y -> sub x (union y).
Lemma sub_union : forall x z,
   (forall y, inc y z -> sub y x)-> sub (union z) x.
```

The union a family of two sets X and Y denoted by $X \cup Y$. An element is in the union if and only if it is in one of the sets. We have $A \subset A \cup B$ and $B \subset A \cup B$.

```
Definition union2 (x y : Set) := union (doubleton x y).

Lemma union2_or : forall x y a, inc a (union2 x y) -> inc a x \/ inc a y.
Lemma union2_first : forall x y a, inc a x -> inc a (union2 x y).
Lemma union2_second : forall x y a, inc a y -> inc a (union2 x y).
Lemma inc_union2_rw : forall a b x,
  inc x (union2 a b) = (inc x a \/ inc x b).
Lemma union2sub_first: forall a b, sub a (union2 a b).
Lemma union2sub_second: forall a b, sub b (union2 a b).
Lemma union2idem:  forall x, union2 x x = x.
Lemma union2comm:  forall x y, union2 x y = union2 y x.
Lemma union2_sub: forall x y,
  sub x y = (union2 x y = y).
```

In some cases (induction on finite sets), one needs to consider the union of a set and a singleton.

```
Definition tack_on x y := union2 x (singleton y).

Lemma tack_on_or : forall x y z : Set, inc z (tack_on x y) ->
  (inc z x \/ z = y).
Lemma tack_on_inc: forall x y z,
  (inc z (tack_on x y) ) = (inc z x \/ z = y).
```

```
Lemma inc_tack_on_x: forall a b, inc a (tack_on b a).
Lemma inc_tack_on_sub: forall a b, sub b (tack_on b a).
Lemma inc_tack_on_y: forall a b y, inc y b -> inc y (tack_on b a).

Lemma tack_on_when_inc: forall x y, inc y x -> tack_on x y = x.
Lemma tack_on_sub: forall x y z, sub x z -> inc y z -> sub (tack_on x y) z.
Lemma tack_on_complement: forall x y, inc y x ->
  x = tack_on (complement x (singleton y)) y.
```

2.10 Module Intersection

Bourbaki defines the *intersection* of a family of sets $(X_\iota)_{\iota\in I}$ as the dual of union. We have $x \in \bigcap_{\iota\in I} X_\iota$ if and only if x is in every element of the family. Fix $\alpha \in I$. Then $x \in \bigcap X_\iota$ if and only if $x \in X_\alpha$ and x is in every element of the family. The construction is independent of α, provided it is in the set of indices I, which must be nonempty. The intersection is a subset of every X_ι. If each X_ι is a subset of E, then the intersection is a subset of E. For this reason Bourbaki defines the intersection, when $I = \emptyset$, to be E. We consider here the case (denoted by $\bigcap X$) where the mapping $\iota \mapsto X_\iota$ is the identity of X, the general case will be studied in the next Chapter.

```
Definition intersection (x : Set) :=
  Zo (rep x) (fun y : Set => forall z : Set, inc z x -> inc y z).

Lemma intersection_inc : forall x a,
 nonempty x -> (forall y, inc y x -> inc a y) -> inc a (intersection x).

Lemma intersection_forall :
 forall x a y, inc a (intersection x) -> inc y x -> inc a y.

Lemma intersection_sub : forall x y, inc y x -> sub (intersection x) y.
```

Intersection of two sets is denoted $X \cap Y$, the properties listed here are obvious.

```
Definition intersection2 (x y : Set) := intersection (doubleton x y).

Lemma intersection2_inc : forall x y a,
  inc a x -> inc a y -> inc a (intersection2 x y).
Lemma intersection2_first :  forall x y a,
  inc a (intersection2 x y) -> inc a x.
Lemma intersection2_second : forall x y a,
  inc a (intersection2 x y) -> inc a y.
Lemma intersection2_both: forall x y a,
  inc a (intersection2 x y) -> (inc a x & inc a y).
Lemma intersection2sub_first: forall a b, sub (intersection2 a b) a.
Lemma intersection2sub_second: forall a b, sub (intersection2 a b) b.
Lemma intersection2idem: forall x, intersection2 x x = x.
Lemma intersection2comm: forall x y, intersection2 x y = intersection2 y x.
Lemma intersection2_sub: forall x y,
  sub x y = (intersection2 x y = x).
```

2.11 Module Transposition

This module is not used for the Bourbaki Project. It has been withdrawn in version 3.

We define here a function T_{ija} that maps i to j, j to i and everything else to a. All lemmas given here are obvious.

```
Definition create (i j a : Set) := Yo (a = i) j (Yo (a = j) i a).

Lemma not_i_not_j : forall i j a, a <> i -> a <> j -> create i j a = a.
Lemma i_j_j_i : forall i j a, create i j a = create j i a.
Lemma i_j_i : forall i j, create i j i = j.
Lemma i_j_j : forall i j, create i j j = i.
Lemma i_i_a : forall i a, create i i a = a.
Lemma surj : forall i j a, exists b, create i j b = a.
Lemma involutive : forall i j a, create i j (create i j a) = a.
Lemma inj : forall i j a b, create i j a = create i j b -> a = b.
```

2.12 Module Bounded

This module is not used for the Bourbaki Project. It has been withdrawn in version 3, after changing definition of the cartesian product.

Let p be a predicate and x a set. Assume $p(y)$ is true if and only if $y \in x$. We say that p satisfies the axioms if there is such a set x. This set is obviously unique, and will be denoted by *create p*.

```
Definition property (p : EP) (x : Set) :=
  forall y : Set, (p y -> inc y x & inc y x -> p y).
Definition axioms (p : EP) := ex (property p).
Definition create (p : EP) := choose (property p).
```

If p satisfies the axioms then y is in *create p* if and only if $p(y)$. If p is bounded (we can consider different cases) then p satisfies the axioms.

```
Lemma lem1 : forall (p : EP) (y : Set), axioms p -> inc y (create p) -> p y.
Lemma lem2 : forall (p : EP) (y : Set), axioms p -> p y -> inc y (create p).
Lemma inc_create : forall (p : EP) y, axioms p -> inc y (create p) = p y.

Lemma criterion : forall p : EP,
 ex (fun x : Set => forall y : Set, p y -> inc y x) -> axioms p.

Lemma trans_criterion :
  forall (p : EP) (f : EE) (x : Set),
    (forall y : Set, p y -> ex (fun z : Set =>  (inc z x) & (f z = y))) ->
    axioms p.

Lemma little_criterion :
  forall (p : EP) (x : Set) (f : x -> Set),
    (forall y : Set, p y -> exists a : x, f a = y) -> axioms p.
```

2.13 Module Cartesian

Note: Implementation has changed in Version 3, for the old definition, see section 9.3.

The *cartesian product* A × B of two sets A and B is the set of all pairs z such that $\mathrm{pr}_1 z \in$ A and $\mathrm{pr}_2 z \in$ B. It is the union (for $x \in$ A) of the sets B_x of all (x, y) for $y \in$ B.

```
Definition product (A B : Set) :=
  union (fun_image A (fun x =>  (fun_image B (fun y => J x y)))).

Lemma product_pr : forall a b u,
    inc u (product a b) -> (is_pair u & inc (P u) a & inc (Q u) b).

Lemma product_inc :  forall a b u,
 is_pair u -> inc (P u) a -> inc (Q u) b -> inc u (product a b).

Lemma product_pair_pr :  forall a b x y,
  inc (J x y) (product a b) -> (inc x a & inc y b).
Lemma product_pair_inc :  forall a b x y,
  inc x a -> inc y b -> inc (J x y) (product a b).
Lemma inc_product : forall x y z,
  inc x (product y z) = (is_pair x & inc (P x) y & inc (Q x) z).
Lemma pair_in_product: forall a b c, inc a (product b c) -> is_pair a.
```

A product is empty if and only one factor is empty. This is Proposition 2 [2, p. 75].

```
Lemma empty_product1: forall y,   product emptyset y = emptyset.
Lemma empty_product2: forall x,   product x emptyset = emptyset.
Lemma empty_product_pr: forall x y,
   product x y = emptyset -> (x = emptyset \/ y= emptyset).
```

The product A × B is increasing in A and B, strictly if the other argument is non empty. This is Proposition 1 [2, p. 74].

```
Lemma product_monotone_left: forall x x' y,
  sub x x' -> sub (product x y) (product x' y).
Lemma product_monotone_right: forall x y y',
  sub y y' -> sub (product x y) (product x y').
Lemma product_monotone: forall x x' y y',
  sub x x' -> sub y y' -> sub (product x y) (product x' y').
Lemma product_monotone_left2: forall x x' y, nonempty y ->
  sub (product x y) (product x' y) -> sub x x'.
Lemma product_monotone_right2: forall x y y', nonempty x ->
  sub (product x y) (product x y') -> sub y y'.
```

2.14 Module Back

This module is not used for the Bourbaki Project. It has been withdrawn in version 2.

We define *RBdefault* as a function of x, z and d, where d is of type z. The value is x if $x \in z$ and $\mathscr{R}d$ otherwise. In any case, this is an element of z.

```
Definition RBdefault x z (d:z) :=
```

```
   Yo (inc x z) x (Ro d).

Lemma RBdefault_in :
  forall x z (d:z), inc x z -> RBdefault x d = x.
Lemma RBdefault_out : forall x z (d:z),
  ~(inc x z) -> RBdefault x d = (Ro d).
Lemma inc_RBdefault : forall x z (d:z),
   inc (RBdefault x d) z.
```

We define now *Bdefault*. If y is the value of $RBdefault(x,z,d)$ we know $y \in z$. Applying $\mathscr{B}$ gives t of type z. If $x \in z$ then $\mathscr{R}t = x$, otherwise $t = d$. If $x = \mathscr{R}x'$, where x' is of type z then $t = x'$.

```
Definition Bdefault x z (d:z) : z :=
  Bo (inc_RBdefault x d).

Lemma R_Bdefault_in : forall x z (d:z),
  inc x z -> Ro (Bdefault x d) = x.

Lemma Bdefault_out : forall x z (d:z),
  ~(inc x z) -> (Bdefault x d) = d.

Lemma Bdefault_R : forall z (y d:z),
  Bdefault (Ro y) d = y.
```

Now the definition of *Bnat*, as a particular case where $z = \mathbb{N}$, and $d = 0$. To any set x we associate an integer n (an object of type *nat*). If $x \in \mathbb{N}$, then $\mathscr{R}n = x$, otherwise $n = 0$.

```
Definition Bnat x := Bdefault x (z:=nat) 0.

Lemma R_Bnat : forall x, inc x nat ->
  Ro (Bnat x) = x.

Lemma Bnat_out : forall x, ~(inc x nat) ->
   Bnat x = 0.

Lemma Bnat_R : forall (i:nat), Bnat (Ro i) = i.
```

Note. In the second part of this document, we shall define *Bnat* as the set of natural integers.

Chapter 3

Functions

We describe in this Chapter the file *jfunc.v* obtained from the file *func.v* by Carlos Simpson, by removing definitions that seemed useless for the Bourbaki project[1]. The module defining equivalence relations will be described in Chapter 6.

3.1 Module Function

A *graph* is a set of pairs. The *domain* and *range* are the images of the first and second projection. A set f satisfies the *fgraph* property if it is a graph and if the first projection is injective. This means that if $(a,b) \in f$ and $(a,b') \in f$ then $b = b'$ (claim that will be proved in the next Chapter).

```
Definition is_graph r := forall y, inc y r -> is_pair y.
Definition domain f  := fun_image f P.
Definition range f := fun_image f Q.
Definition fgraph f :=
  is_graph f & (forall x y, inc x f -> inc y f -> P x = P y -> x = y).
```

The domain and range are characterized by the following two lemmas.

```
Lemma is_graph_fgraph : forall f, fgraph f -> is_graph f.
Lemma fgraph_pr: forall f x y y',
  fgraph f -> inc (J x y) f -> inc (J x y') f -> y = y'.
Lemma domain_pr: forall r x,
  is_graph r -> inc x (domain r) = (exists y, inc (J x y) r).

Lemma range_pr: forall r y,
  is_graph r -> inc y (range r) = (exists x, inc (J x y) r).
```

These lemmas are obvious from the definitions and the fact that a functional graph is a graph.

```
Lemma inc_pr1_domain : forall f x,
  fgraph f -> inc x f -> inc (P x) (domain f).

Lemma inc_pr2_range : forall f x,
  fgraph f -> inc x f -> inc (Q x) (range f).
```

[1] In version 2, the modules have been moved to the file set1

If g is a non-empty graph, its domain is non-empty. In the first lemma given here, we do no say that that g is a graph. In fact, if x is in the domain of g, there need not exist a y such that $(x, y) \in g$, we just say that there exists $z \in g$ with $x = \mathrm{pr}_1 z$. In the second lemma, we add the assumption that g is a graph, and we know that there is at least one y, namely $\mathcal{V}(x, g)$.

```
Lemma nonempty_domain: forall g,
  nonempty g -> nonempty (domain g).

Lemma fdefined_lem : forall f x,
  fgraph f -> inc x (domain f) -> inc (J x (V x f)) f.
```

The first lemma says that if x is in the graph, then x is a pair whose second component is $\mathcal{V}(\mathrm{pr}_1 x, f)$. The second lemma says that y is in the range if and only if it is $\mathcal{V}(x, f)$ for some x in the domain. Other two lemmas are trivial consequences.

```
Lemma in_graph_V : forall f x,
  fgraph f -> inc x f -> x = J (P x) (V (P x) f).

Lemma frange_inc_rw : forall f y,
  fgraph f -> inc y (range f) = (exists x, inc x (domain f) & y = V x f).

Lemma pr2_V : forall f x,
  fgraph f -> inc x f -> Q x = V (P x) f.

Lemma inc_V_range:  forall f x,
  fgraph f -> inc x (domain f) -> inc (V x f) (range f).
```

Assume that g is a functional graph, and $f \subset g$. Then f is a functional graph, its domain and range are subsets of the domain and range of g; its evaluation function is the same. There is a converse: if we have two functional graphs, if the domain of f is a part of the domain of g, and if the evaluation function is the same on the domain of f, then f is a subset of g. From this we deduce an extensionality property.

```
Lemma sub_axioms : forall f g, fgraph g -> sub f g -> fgraph f.
Lemma sub_domain : forall f g, sub f g -> sub (domain f) (domain g).
Lemma sub_range : forall f g, sub f g -> sub (range f) (range g).
Lemma sub_ev: forall f g x,
  fgraph g -> sub f g -> inc x (domain f) -> V x f = V x g.
Lemma function_sub : forall f g,
  fgraph f -> fgraph g ->
  sub  (domain f) (domain g) ->
  (forall x, inc x (domain f) -> V x f = V x g) -> sub f g.
Lemma function_extensionality: forall f g,
  fgraph f -> fgraph g -> domain f = domain g ->
  (forall x, inc x (domain f) -> V x f = V x g) -> f = g.
```

¶ Inverse image of a set a by a graph f, denoted $\overset{-1}{f}\langle a\rangle$ or simply $f^{-1}\langle a\rangle$. This is a part of the domain, characterized by the property that $x \in f^{-1}\langle a\rangle$ if and only if $\mathcal{V}(x, f) \in a$.

```
Definition inverse_image (a f : Set) :=
  Zo (domain f) (fun x  => inc (V x f) a).

Lemma inverse_image_sub :   forall a f,
  sub (inverse_image a f) (domain f).
```

```
Lemma inverse_image_inc :  forall a f x,
  inc x (domain f) -> inc (V x f) a -> inc x (inverse_image a f).
Lemma inverse_image_pr : forall a f x,
  inc x (inverse_image a f) -> inc (V x f) a.
```

Consider now a function f, and a set x. The set of all pairs $(a, f(a))$ for $a \in x$ will be denoted by $\mathscr{L}_x f$. This is a functional graph; its domain is x, and its evaluation function is f.

```
Definition function_create (x : Set) (p : E) :=
  fun_image x (fun y => J y (p y)).
Notation L := function_create.

Lemma create_axioms : forall p x, fgraph (L x p).
Lemma create_domain : forall x p, domain (L x p) = x.
Lemma create_V_rewrite :  forall x p y,
  inc y x -> V y (L x p) = p y.
```

The range of $\mathscr{L}_x f$ is the image $f\langle x\rangle$ (according to Section 2.7; on page 44 we shall define $g\langle x\rangle$ where g is a graph). There are some other useful properties.

If v is a graph with domain x and evaluation function f, then $v = \mathscr{L}_x f$. We have $\mathscr{L}_x f = \mathscr{L}_y g$ if $x = y$, and f and g agree on x.

```
Lemma create_range :  forall p x,
  range (L x p) = fun_image x p.
Lemma create_create : forall  a f,
  L a (fun x => V x (L a f)) = L a f.
Lemma inc_create_range: forall sf f a,
  inc a (range (L sf f))  = exists b, inc b sf & f b = a.
Lemma create_V_out : forall x f y,
  ~inc y x -> V y (L x f) = emptyset.
Lemma create_recovers :  forall f,
  fgraph f -> L (domain f) (fun x : Set => V x f) = f.
Lemma function_extensionality1 : forall a b f g,
  a = b ->  (forall x, inc x a -> f x = g x) ->
   L a f = L b g.
```

¶ We denote by $g \circ f$ the *composition* of the two functions. It maps x to $g(f(x))$. In the case of graphs, the evaluation function is $\mathscr{V}(\mathscr{V}(x, f), g)$; note that the order is reversed. The domain is the set of all x in the domain of f that are mapped to the domain of g, it is the inverse image of the domain of g by f. We do not like this definition, thus introduce an alternate one, that agrees if functions are composable. Note that the last lemma makes no assumptions on f and g. The easy case is when the two objects are composable (in particular, they are functions). In this case the domain of $g \circ f$ is the domain of f.

```
Definition fcomposable (f g : Set) :=
  fgraph f & fgraph g & sub (range g) (domain f).

Definition fcompose (f g : Set) :=
  L (inverse_image (domain f) g) (fun y => V (V y g) f).
Definition gcompose g f := L(domain f)  (fun y => V (V y f) g).

Lemma fcompose_axioms : forall f g, fgraph (fcompose f g).
Lemma fcompose_domain :  forall f g,
  domain (fcompose f g) = inverse_image (domain f) g.
```

```
Lemma fcompose_range:  forall f g, fgraph f ->
  sub (range (fcompose f g)) (range f).
Lemma fcomposable_domain : forall f g,
  fcomposable f g -> domain (fcompose f g) = domain g.
Lemma alternate_compose: forall g f,
  fcomposable g f -> gcompose g f = fcompose g f.
Lemma fcompose_ev : forall x f g,
  inc x (domain (fcompose f g)) -> V x (fcompose f g) = V (V x g) f.
```

An interesting function is the *identity*: it maps everything on itself. More properties will be given later.

```
Definition identity (x : Set) := L x (fun y : Set => y).

Lemma identity_axioms : forall x, fgraph (identity x).
Lemma identity_domain : forall x, domain (identity x) = x.
Lemma identity_ev : forall x a, inc x a -> V x (identity a) = x.
```

Given two sets f and x, one can consider the set of all $y \in f$ satisfying $\mathrm{pr}_1\, y \in x$. This makes sense if f is a graph, it is called the *restriction* of f to x. In fact, since this is a subset of f, it is a functional graph whenever f is. Its domain is the intersection of f and x. On the restriction domain the function takes the same value as the restriction.

```
Definition restr f x :=
  Zo f (fun y=> inc (P y) x).

Lemma restr_inc_rw : forall f x y,
  inc y (restr f x) = (inc y f & inc (P y) x).
Lemma restr_sub : forall f x,
   sub (restr f x) f.
Lemma restr_axioms : forall f x,
  fgraph f -> fgraph (restr f x).

Lemma restr_domain : forall f x,
  fgraph f ->  domain (restr f x) = intersection2 (domain f) x.
Lemma restr_domain1 : forall f x,
  fgraph f -> sub x (domain f) -> domain (restr f x) = x.
Lemma restr_ev : forall f u x,
  fgraph f -> sub u (domain f) -> inc x u ->
    V x (restr f u) = V x f.

Lemma function_sub_V : forall f g x,
  fgraph g -> defined f x -> sub f g ->
    V x f = V x g.

Lemma function_sub_eq : forall r s,
  fgraph r -> fgraph s -> sub r s ->
    sub (domain s) (domain r) -> r = s.

Lemma restr_to_domain : forall f g,
  fgraph f -> fgraph g -> sub f g ->  restr g (domain f) = f.
```

We say that f is a restriction of g if there is a set x such that f is the restriction of g to x. This is the same as saying that f is a subset of g.

```
Definition is_restriction (f g :Set) :=
```

```
  fgraph g &  exists x, f = restr g x.

Lemma is_restriction_pr: forall f g,
  is_restriction f g =   (fgraph f & fgraph g & sub f g).
```

The union of functional graphs is a graph, provided that some compatibility condition holds. The domain is the union of the domains. The range is the union of the ranges. We consider the special case of a union of a graph and the singleton $\{(x, y)\}$.

```
Lemma domain_union : forall z, domain (union z) =
  union (fun_image z domain).

Lemma domain_tack_on : forall f x y,
  domain (tack_on f (J x y)) = tack_on (domain f) x.

Lemma range_union : forall z, range (union z) =
  union (fun_image z range).

Lemma range_tack_on : forall f x y,
  range (tack_on f (J x y)) = tack_on (range f) y.

Lemma function_tack_on_axioms : forall f x y,
  fgraph f -> ~inc x (domain f) ->
  fgraph (tack_on f (J x y)).
```

Given a function that takes an argument of type x, we know how to convert it to a function defined on the set x. We can then take its graph.

```
Definition tcreate (x:Set) (f:x->Set) :=
  L x (fun y => (Yy (fun (hyp : inc y x) => f (Bo hyp)) emptyset)).

Lemma tcreate_value_type : forall x (f:x->Set) y,
  V (Ro y) (tcreate f) = f y.

Lemma tcreate_value_inc : forall x (f:x->Set) y (hyp : inc y x),
  V y (tcreate f) = f (Bo hyp).
Lemma domain_tcreate : forall x (f:x->Set), domain (tcreate f) = x.
```

3.2 Module FunctionSet

This module is not used for the Bourbaki Project. It has been withdrawn in version 3.

We say that u is in $\mathrm{F}(a, f)$ if u is a functional graph, its domain is a and for every x, the value $\mathcal{V}(x, u)$ is in $f(x)$. In section 2.13 we have defined the record $\mathrm{R}(a, f)$. It is immediate that $u \subset \mathrm{R}(a, f)$; this has as consequence that the property is bounded.

```
Definition in_function_set (a : Set) (f : EE) (u : Set) :=
  fgraph u
  & domain u = a
  & (forall y, inc y a -> inc (V y u) (f y)).

Lemma in_fs_sub_record :  forall a f u,
 in_function_set a f u -> sub u (record a f).
```

```
Lemma in_fs_eq_L :  forall a f u,
 in_function_set a f u -> u = L a (fun y : Set => V y u).

Lemma in_fs_for_L :  forall a g v,
 (forall y, inc y a -> inc (v y) (f y)) ->
 in_function_set a f (L a v).

Lemma in_fs_bounded : forall a f,
 Bounded.axioms (in_function_set a f).
```

Since the property is bounded, there exists a set, denoted F(a, f) above. It is a subset of $\mathfrak{P}$(R(a, f)).

```
Definition function_set (a : Set) (f : EE) :=
  Bounded.create (in_function_set a f).

Lemma function_set_iff :  forall  a f u
 inc u (function_set a f) <-> in_function_set a f u.

Lemma function_set_sub_powerset_record :  forall a f,
 sub (function_set a f) (powerset (record a f)).

Lemma function_set_pr : forall a f u,
    inc u (function_set a f) ->
    (in_function_set a f u & fgraph u & domain u = a
      & (forall y, inc y a -> inc (V y u) (f y))).

Lemma function_set_inc :  forall a f u,
    fgraph u -> domain u = a ->
    (forall y, inc y a -> inc (V y u) (f y)) -> inc u (function_set a f).

Lemma in_function_set_inc :  forall a f u,
 in_function_set a f u -> inc u (function_set a f).
```

3.3 Module Notation

Note. This module has been withdrawn in Version 3. The content has been moved to the file algebra3.

Bourbaki says: a correspondence f = (F, A, B) is said to be function if its graph F is a functional graph and if its source A is equal to its domain pr_1F [2, p. 81], and has statements of the form: let f be a mapping of A into B, if f is injective and if A $\neq \emptyset$ then f has a left-inverse. The important point is that a function contains three items (source, graph, etc., and has some properties as A = pr_1F). In Coq this means that *source f = P (graph f)* is true for every function f. We do not require *graph f = P f*. In a first implementation, the graph was defined by the function *graphC* below. Then we changed our mind and associated a record to it; the trouble is then that we cannot consider the set of all functions, but must use instead the set of all triples (F, A, B) associated to the function.

Consider now the following puzzle. We say that a group is a tuple $(E, +, -, 0)$ with some properties and that a ring is a tuple $(E, +, -, 0, *, 1)$ and that a field is a tuple $(E, +, -, 0, *, 1, /)$. How can we arrange the data structure so that properties true for a group become true for a field?

One solution is the following: we say that a group is tuple of equalities ($u = \mathrm{E}, p = +, m = -, z = 0$), where u, p, m, z are some constant names (we will use character strings in what follows). The order becomes irrelevant, and we may have additional, useless, elements. Instead of $u = \mathrm{E}$ we use the pair (u, E). In other words, a group is a finite functional graph.

A *notation* is a functional graph, whose domain is *string*. A finite functional graph is defined by the two constructors *stop* and *denote*. Here *stop* is the constant function that associates the empty set to every value and *denote a b f* is the function similar to f, but it associates b to a.

```
Definition is_notation f :=
  fgraph f & domain f = string.

Definition stop := L string (fun s => emptyset).

Definition denote str obj old :=
  L string (fun s => (Yo (s = str) obj (V s old))).
```

The following four lemmas explain how to use the notation mechanism.

```
Lemma is_notation_stop : is_notation stop.
Lemma is_notation_denote : forall str obj old,
  is_notation old -> is_notation (denote str obj old).

Lemma V_stop : forall x, V x stop = emptyset.
Lemma V_denote_new : forall str obj old x,
  x = str -> inc x string -> V x (denote str obj old) = obj.

Lemma V_denote_old : forall str obj old x,
  ~x=str -> inc x string -> V x (denote str obj old) = V x old.
```

We define here some commonly used fields.

```
Definition Underlying := Ro "Underlying".
Definition Source  := Ro "Source".
Definition Target  := Ro "Target".
Definition Graph  := Ro "Graph".
Definition Arrow  := Ro "Arrow".

Definition Ul (x : E) := V Underlying x.
Definition sourceC (x : E) := V Source x.
Definition graphC (x : E) := V Graph x.
Definition targetC (x : E) := V Target x.
Definition arrowC x := V Arrow x.
```

This may be used later one when defining unary and binary operations on a set.

```
Definition unary (x:Set) (f:EE) := L x f.

Lemma V_unary : forall x f a, inc a x ->
  V a (unary x f) = f a.

Definition binary (x:Set) (f:EEE) :=
  L x (fun a => (L x (fun b => f b a))).
```

```
Lemma V_V_binary : forall x f a b,
  inc a x -> inc b x -> V a (V b (binary x f)) = f a b.
```

3.4 Module Universe

This module has been withdrawn in the second edition.

A *universe* is some big set. The definition here says that if u is a universe then it must contain many sets.

```
Definition axioms u :=
  (forall x y, inc x u -> inc y x -> inc y u) &
  (forall x (f:x->Set), inc x u -> (sub (IM f) u) -> inc (IM f) u) &
  (forall x, inc x u -> inc (union x) u) &
  (forall x, inc x u -> inc (powerset x) u) &
  inc nat u &
  inc string u.
```

We give here a list of properties of a universe.

```
Lemma inc_trans_u : forall x u, axioms u ->
  (exists y, (inc x y & inc y u)) -> inc x u.

Lemma inc_powerset_u : forall x u, axioms u ->  inc x u ->
   inc (powerset x) u.

Lemma inc_nat_u : forall u, axioms u -> inc nat u.
Lemma inc_R_nat_u : forall (n:nat) u, axioms u -> inc (Ro n) u.
Lemma inc_prop_u : forall u, axioms u -> inc Prop u.
Lemma inc_R_a_prop_u : forall u (p:Prop), axioms u ->  inc (Ro p) u.
Lemma inc_a_prop_u : forall u (p:Prop), axioms u -> inc p u.
Lemma inc_proof_u : forall u (p:Prop) (t:p), axioms u -> inc (Ro t) u.
Lemma inc_string_u : forall u, axioms u -> inc string u.
Lemma inc_subset_u : forall x u, axioms u ->
  (exists y, (inc y u & sub x y)) -> inc x u.
Lemma inc_emptyset_u : forall u, axioms u -> inc emptyset u.
Lemma inc_IM_u : forall x (f:x->Set) u,
  axioms u -> inc x u -> sub (IM f) u ->
  inc (IM f) u.

Definition doubleton_step :forall (x y:Set) (n:nat), Set.
intros. induction n. exact x. exact y.
Defined.

Lemma IM_doubleton_step: forall x y,
IM (doubleton_step x y) = (doubleton x y).

Lemma inc_doubleton_u : forall x y u,
  axioms u -> inc x u -> inc y u -> inc (doubleton x y) u.
Lemma inc_singleton_u : forall x u,
Lemma inc_pair_u : forall x y u,
  axioms u -> inc x u -> inc y u -> inc (pair x y) u.
Lemma sub_u : forall x u, axioms u -> inc x u -> sub x u.
Lemma inc_function_create_u : forall x f u,
```

```
   axioms u -> inc x u ->
  (forall y, inc y x -> inc (f y) u) ->
  inc (L x f) u.
Lemma inc_function_tcreate_u : forall x (f:x->Set) u,
  axioms u -> inc x u ->
  (forall y, inc (f y) u) ->
  inc (tcreate f) u.
Lemma inc_pr1_of_pair_u : forall u x,
  axioms u -> (exists y, inc (pair x y) u) -> inc x u.
Lemma inc_pr2_of_pair_u : forall u y,
  axioms u -> (exists x, inc (pair x y) u) -> inc y u.
Lemma inc_V_u : forall f x u,
  axioms u -> inc f u -> inc x u -> inc (V x f) u.
Lemma inc_denote_u : forall s x a u,
  axioms u -> inc a u -> inc s string -> inc x u ->
Lemma inc_binary_u : forall x f u,
  axioms u -> inc x u ->
  (forall y z, inc y x -> inc z x -> inc (f y z) u) ->
  inc (binary x f) u.
Lemma inc_stop_u : forall u,
   axioms u -> inc stop u.
```

Chapter 4

Correspondences

From now on, we follow Bourbaki as closely as possible. The series "Elements of mathematics" is divided in 9 books, the first one is called "Theory of sets". This book is divided into four chapters, the second one is "Theory of sets". This chapter is divided into 6 sections; we implement here section 3 "Correspondences". When we talk about Proposition 1, this is to be understood as Proposition 1 of [2] of the current section (i.e., the current Chapter of this report).

We consider here some properties of sections 1 (Collectivizing relations) and 2 (Ordered pairs) not implemented by Carlos Simpson in [2].

If $a \subset x$ and $b \subset x$, then $x \setminus a \subset x \setminus b$ if and only if $b \subset a$.

```
Lemma complement_monotone : forall a b x,
  sub a x -> sub b x -> (sub a b = sub (complement x b) (complement x a)).
```

A property P(y) is collectivizing if there is a set x such that P(y) is equivalent to $y \in x$. There are properties that are not collectivizing, for instance $y \notin y$. The second lemma says that there is a set containing no set but there is no set containing all sets.

```
Lemma not_collectivizing_notin:
  ~ (exists z, forall y, inc y z = not (inc y y)).
Lemma collectivizing_special :
  (exists x, forall y, ~ (inc y x)) &  ~ (exists x, forall y, inc y x).
```

Additional properties of the empty set. We have $\emptyset \subset x$ for every x, but the converse is true only if x is the empty set. Since $x \in \emptyset$ is absurd, everything can be deduced from it.

```
Lemma emptyset_pr: forall x,  inc x emptyset ->  False.
Lemma emptyset_pra: forall x (p: EP),  inc x emptyset -> (p x).
Lemma sub_emptyset : forall x,  sub x emptyset = (x = emptyset).
```

4.1 Graphs and correspondences

A graph r is a set of pairs; if the pair (x, y) is an element of r we say that x and y are related by r. This will be used essentially when r is the graph of a relation.

```
Definition related r x y := inc (pair x y) r.
```

The next theorem is Proposition 1 in [2, p. 76]; it claims existence and uniqueness of two sets denoted by $\mathrm{pr}_1\langle r\rangle$ and $\mathrm{pr}_2\langle r\rangle$. The notation $\mathrm{pr}_1\langle r\rangle$ is defined in section 2.7; it is the domain of r.

```
Theorem range_domain_exists: forall r,
  is_graph r ->
  (exists_unique (fun a=> (forall x, inc x a = (exists y, inc (J x y) r))) &
    exists_unique (fun b=> (forall y, inc y b  = (exists x, inc (J x y) r)))).
```

A graph is a subset of the product of the domain by the range. A graph is empty if and only if its domain or range is empty. A functional graph is a graph.

```
Lemma sub_graph_prod: forall r, is_graph r ->
  sub r (product (domain r)(range r)).
Lemma empty_graph1: forall r, is_graph r ->
  (domain r = emptyset) = (r = emptyset).
Lemma empty_graph2: forall r, is_graph r ->
  (range r = emptyset) =  (r = emptyset).
Lemma graph_fgraph : forall f, fgraph f -> is_graph f.
```

The emptyset is a functional graph, with empty range and domain.

```
Lemma emptyset_is_graph: is_graph (emptyset).
Lemma range_emptyset: range emptyset = emptyset.
Lemma domain_emptyset: domain emptyset = emptyset.
Lemma fgraph_emptyset: fgraph emptyset.
```

A product $x \times y$ is a graph. The domain is x, the range is y. Note that, if one set is empty, then the product is empty, see above. It is a functional graph if the range is a singleton.

```
Lemma product_is_graph: forall x y,
  is_graph (product x y).
Lemma product_related: forall x y a b,
  related (product x y) a b = (inc a x & inc b y).
Lemma domain_product: forall x y,
  nonempty y-> domain (product x y) = x.
Lemma range_product: forall x y,
  nonempty x -> range (product x y) = y.
Lemma constant_function_p1: forall x y,
  fgraph (product x (singleton y)).
```

The *diagonal* of x, denoted Δ_x, is the set of all pairs (a, a), with $a \in x$. This is the graph of the identity function on x, domain and range being x.

```
Definition diagonal x := Zo (product x x)(fun y=> P y = Q y).

Lemma diagonal_is_identity: forall x, diagonal x = identity x.
Lemma inc_diagonal: forall x u,
  inc u (diagonal x) = (is_pair u & inc (P u) x & P u = Q u).
Lemma inc_pair_diagonal: forall x u v,
  inc (J u v) (diagonal x)  = (inc u x & u = v).
Lemma diagonal_related: forall x u v,
  related (diagonal x) u v = (inc u x & u = v).
```

```
Lemma fgraph_diagonal:  forall x, fgraph (diagonal x).
Lemma diagonal_is_graph:  forall x, is_graph (diagonal x).
Lemma domain_diagonal: forall x, domain (diagonal x) = x.
Lemma range_diagonal: forall x, range (diagonal x) = x.
Lemma V_diagonal: forall a x, inc a x -> V a (diagonal x) = a.
```

For Bourbaki, a *correspondence* between A and B is a triple $\Gamma = (G, A, B)$ where the domain of G is a subset of A and the range is a subset of B. We use here a record γ, and define two conversion functions $\Gamma = \mathrm{cv}(\gamma)$, and $\gamma = \mathrm{icv}(\Gamma)$.

```
Record correspondenceC:Type :=
  corresp{ source:Set; target:Set; graph :Set }.
Definition corr_value (x:correspondenceC):=
  J(graph x) (J (source x) (target x)).
Definition inv_corr_value z := corresp(P (Q z)) (Q (Q z)) (P z).
```

We have $\mathrm{icv}(\mathrm{cv}(\gamma)) = \gamma$ for any correspondence γ, and $\gamma = \gamma'$ if and only if $\mathrm{cv}(\gamma) = \mathrm{cv}(\gamma')$.

```
Lemma corr_propc: forall f,
  corresp(source f) (target f) (graph f) =f.
Lemma inv_corr_value_pr: forall z, inv_corr_value (corr_value z) = z.

Lemma correspondence_extensionality1: forall m m',
  (corr_value m = corr_value m') = (m = m').
```

A data structure γ with values G, A and B is called a correspondence if G is a graph, whose domain is a subset of A and whose range is a subset of B. We give a similar definition for a triple.

```
Definition corr_axiom s t g:=
  is_graph g & sub (domain g) s & sub (range g) t.
Definition corr_axiom1 x :=
  is_pair x & is_pair (Q x) &  corr_axiom (P (Q x)) (Q (Q x)) (P x).
Definition is_correspondence f :=
  corr_axiom (source f) (target f) (graph f).
```

We list here the basic properties of correspondences. The non-trivial one is that γ is a correspondence if and only if $G \subset A \times BB$.

```
Lemma corr_create: forall s t g,
  corr_axiom s t g -> is_correspondence(corresp s t g).
Lemma corr_props: forall f,
  is_correspondence f -> corr_axiom(source f) (target f)(graph f).
Lemma is_graph_correspondence: forall g,
  is_correspondence g -> is_graph(graph g).
Lemma range_correspondence: forall g,
  is_correspondence g -> sub (range (graph g)) (target g).
Lemma domain_correspondence: forall g,
  is_correspondence g -> sub (domain (graph g)) (source g).

Lemma corr_axiom_rw: forall s t g,
  corr_axiom s t g = sub g (product s t).
```

A triple (G, A, B) is a correspondence if and only if $G \in \mathfrak{P}(A \times B)$, but Bourbaki defines the powerset only later. From this, we deduce that the set of all correspondences between A and

B is $\mathfrak{P}(A \times B) \times \{A\} \times \{B\}$. If E is this set, we show that $z \in E$ if and only if z is the value of a correspondence Γ from A to B (and such a Γ is unique).

```
Definition set_of_correspondences (x y:Set) :=
  product(powerset (product x y))
  (product (singleton x) (singleton y)).
Definition sof_value x y z := corresp x y (P z).

Lemma corr_propa: forall x y z,
   corr_axiom x y z = inc z (powerset (product x y)).
Lemma set_of_correspondences_prop: forall x y z,
  inc z (set_of_correspondences x y) =
  (corr_axiom1 z & P (Q z) = x & Q (Q z) = y).
Lemma set_of_correspondences_propa: forall f,
  is_correspondence f ->
  inc (corr_value f) (set_of_correspondences (source f) (target f)).
Lemma sof_value_pra:  forall x y z,
  inc z (set_of_correspondences x y) ->
    (is_correspondence (sof_value x y z) &
    source (sof_value x y z) = x &
    target (sof_value x y z)  = y &
    corr_value (sof_value x y z) =z).
Lemma set_of_correspondences_propb: forall x y z,
  inc z (set_of_correspondences x y) ->
  exists f,  is_correspondence f & source f = x & target f = y & corr_value f=z.
```

Given a function $f : a \to b$, we construct $\mathscr{L} f$, the associated correspondence.

```
Definition gacreate (a b:Set) (f:a->b) := IM (fun y:a => J (Ro y) (Ro (f y))).
Definition acreate (a b:Set) (f:a->b)  := corresp a b (gacreate f).

Lemma acreate_axioms: forall (a b:Set) (f:a->b),
  is_correspondence (acreate f).
Lemma source_acreate : forall (a b :Set)(f:a->b),  source (acreate f) = a.
Lemma target_acreate : forall (a b:Set) (f:a->b),  target (acreate f) = b.
```

Restriction of f to x is defined elsewhere. In some cases, it is important to change the order of arguments.

```
Definition restriction_graph f r := restr r f.

Lemma restriction_graph_pr: forall x r y,
  (inc y (restriction_graph x r) = (inc y r & inc (P y) x)).

Lemma restricted_graph_is_graph: forall x r,
  is_graph r -> is_graph (restriction_graph x r).
```

¶ Direct image of a set by a functional object. This will be denoted by $f\langle x\rangle$. In the first definition f is a graph, and we consider all elements y for which there is a $z \in x$ such that $(z, y) \in f$. In the second definition, f is a correspondence, and we consider the image by its graph. In the last definition, f is a correspondence, and we take the image of the source (the case where f is a mapping has been considered in Section 2.7).

```
Definition image_by_graph f u:=
```

```
  Zo(range f)(fun y=>exists x, inc x u & inc (J x y) f).

Definition image_by_fun f u :=
  image_by_graph(graph f) u.

Definition image_of_fun f :=
  image_by_graph (graph f) (source f).
```

We give now some basic properties. The image is a part of the range; it is the full range if we consider the full domain. The image of a subset x of the domain is empty if and only if x is empty. Proposition 2 in [2, p. 77] says that the image functor is increasing (we use here the term "functor" rather than function, since it is a mapping without graph).

```
Lemma image_by_graph_pr: forall u r y,
  inc y (image_by_graph r u) = exists x, (inc x u & inc (J x y) r).
Lemma sub_image_by_graph: forall u r,
  sub (image_by_graph r u)  (range r).
Lemma image_by_graph_domain: forall r, is_graph r ->
  image_by_graph r (domain r) = range r.
Lemma image_by_emptyset: forall r,
  image_by_graph r emptyset =  emptyset.
Lemma image_by_nonemptyset: forall u r,
  is_graph r -> nonempty u -> sub u (domain r)
  -> nonempty (image_by_graph r u).
Theorem image_by_increasing: forall u u' r,
  is_graph r -> sub u u' -> sub (image_by_graph r u) (image_by_graph r u').
Lemma image_of_large: forall u r,  is_graph r ->
  sub (domain r) u -> image_by_graph r u = range r.
```

Given a graph r and an element x, the set of all y in r whose first projection is x is called the *cut*. This is $r\langle\{x\}\rangle$. If f is a correspondence, the notation $\mathrm{G}(f)\langle\{x\}\rangle$ is sometimes simplified to $f\langle\{x\}\rangle$ or $f(x)$ (this last notation is ambiguous, since it denotes also the value of f at x).

```
Definition im_singleton r x := image_by_graph r (singleton x).

Lemma im_singleton_pr: forall r x y,
  inc y (im_singleton r x) = inc (J x y) r.
Lemma im_singleton_inclusion: forall r r', is_graph r -> is_graph r' ->
  (forall x, sub (im_singleton r x) (im_singleton r' x)) = sub r r'.
```

4.2 Inverse of a correspondence

The inverse graph of G, denoted by $\overset{-1}{\mathrm{G}}$, or G^{-1} is the set of all pairs (x, y) such that $(y, x) \in \mathrm{G}$. We follow the definition of Bourbaki; he says that this set exists when G is a graph, because it is a subset of the product of the range and domain. We can also consider the image of the mapping $(x, y) \to (y, x)$. Both definitions agree if G is a graph.

```
Definition inverse_graph r :=
  Zo(product(range r)(domain r))
  (fun y=> inc (J (Q y)(P y)) r).

Lemma inverse_graph_alt: forall r, is_graph r ->
  inverse_graph r = fun_image r (fun z => J(Q z) (P z)).
```

Some trivialities to start with.

```
Lemma inverse_graph_is_graph: forall r, is_graph (inverse_graph r).
Lemma inverse_graph_pr: forall r y, is_graph r ->
  inc y (inverse_graph r) = (is_pair y &  inc (J (Q y)(P y)) r).
Lemma inverse_graph_pair: forall r x y,
  inc (J x y) (inverse_graph r) = inc (J y x) r.
Lemma inverse_graph_pr2: forall r x y,
  related (inverse_graph r) y x = related r x y.
```

Taking the inverse swaps range and domain. Taking twice the inverse gives the same graph. The inverse of a product is the product in reverse order. The inverse of the empty set or identity is itself.

```
Lemma inverse_graph_involutive: forall r, is_graph r ->
  inverse_graph (inverse_graph r) = r.
Lemma range_inverse: forall r, is_graph r ->
  range (inverse_graph r) = domain r.
Lemma domain_inverse: forall r, is_graph r ->
  domain (inverse_graph r) = range r.
Lemma inverse_graph_emptyset:
  inverse_graph (emptyset) = emptyset.
Lemma inverse_product: forall x y,
  inverse_graph (product x y) = product y x.
Lemma inverse_diagonal: forall x,
  inverse_graph (diagonal x) = diagonal x.
```

The inverse of the correspondence $\Gamma = (G, A, B)$ is $(\overset{-1}{G}, B, A)$. It is denoted by $\overset{-1}{\Gamma}$. It satisfies some trivial properties.

```
Definition inverse_fun m :=
  corresp(target m) (source m)(inverse_graph (graph m)).

Lemma correspondence_inverse_fun: forall m,
  is_correspondence m -> is_correspondence (inverse_fun m).
Lemma source_inverse: forall m: correspondenceC,
  source(inverse_fun m) = target m.
Lemma target_inverse: forall m: correspondenceC,
  target(inverse_fun m) = source m.
Lemma graph_inverse: forall m: correspondenceC,
  graph(inverse_fun m) = inverse_graph(graph m).
Lemma inverse_fun_involutive: forall m,
  is_correspondence m -> inverse_fun (inverse_fun m) = m.
```

The inverse image by a graph (or correspondence or a function) is the direct image of its inverse. It is denoted by $g^{-1}\langle x\rangle$.

```
Definition inv_image_by_graph r x :=
  image_by_graph (inverse_graph r) x.

Definition inv_image_by_fun r x:=
  inv_image_by_graph(graph r) x.

Lemma inv_image_by_fun_pr: forall r x,
  inv_image_by_fun r x = image_by_fun (inverse_fun r) x.
Lemma inv_image_graph_pr: forall x r y,
  (inc y (inv_image_by_graph r x)) = (exists u, inc u x & inc (J y u) r)).
```

4.3 Composition of two correspondences

The *composition* of two graphs $G_2 \circ G_1$ is the set of all (x, z) for which there is an y such that (x, y) is in the first graph and (y, z) is in the second. (this agrees with the previous definition in good cases). It is a subset of the product of the domain of the first graph and the range of the second. Note: the first graph is G_1, it is the second argument of *compose_graph*. These properties are obvious.[1]

```
Definition compose_graph r' r :=
 Zo(product(domain r)(range r'))(fun w => exists y,
   (inc (J (P w) y) r & inc (J y (Q w)) r')).

Lemma composition_is_graph: forall r r',
  is_graph (compose_graph r r').
Lemma inc_compose: forall r r' x,
  inc x (compose_graph r' r) =
  (is_pair x &( exists y, inc (J (P x) y) r& inc (J y (Q x)) r')).
Lemma compose_related:forall r r' x z,
  (related (compose_graph r' r) x z=
  exists y, related r x y & related r' y z).
Lemma compose_domain1: forall r r',
  sub (domain (compose_graph r' r)) (domain r).
Lemma compose_range1:forall r r',
  sub (range (compose_graph r' r)) (range r').
```

Proposition 3 in [2, p. 79] says $(G' \circ G)^{-1} = G^{-1} \circ (G')^{-1}$.

```
Theorem inverse_compose:forall r r',
  inverse_graph (compose_graph r' r) =
  compose_graph (inverse_graph r)(inverse_graph r').
```

Proposition 4 [2, p. 79] says that graph composition is associative.

```
Theorem composition_associative:forall r r' r'',
  is_graph r -> is_graph r' -> is_graph r'' ->
  compose_graph r'' (compose_graph r' r) =
  compose_graph (compose_graph r'' r') r.
```

Proposition 5 [2, p. 79] says $(G' \circ G)\langle A\rangle = G'\langle G\langle A\rangle\rangle$. We have a characterization of the domain and range of the composition as direct or inverse image of the domain or range. We have an interesting formula $A \subset G^{-1}\langle G\langle A\rangle\rangle$.

```
Theorem image_composition: forall r r' x,
  image_by_graph(compose_graph r' r) x = image_by_graph r' (image_by_graph r x).

Lemma compose_domain:forall r r',
  is_graph r' ->
  domain (compose_graph r' r) = inv_image_by_graph r(domain r').

Lemma compose_range: forall r r',
  is_graph r ->
  range (compose_graph r' r) =   image_by_graph r' (range r).
```

[1] In a previous version, we assumed in some lemmas that r or r' are graphs

```
Lemma inverse_direct_image: forall r x,
  is_graph r -> sub x (domain r) ->
  sub x (inv_image_by_graph r (image_by_graph r x)).

Lemma composition_increasing: forall r r' s s',
  sub r s -> sub r' s' -> sub (compose_graph r' r) (compose_graph s' s).
```

The property that f and f' are two correspondences where the target of f is the source of f' will be called *composableC*. We can write them as $f = (\mathrm{G}, \mathrm{A}, \mathrm{B})$ and $f' = (\mathrm{G}', \mathrm{B}, \mathrm{C})$. We define the *composition* $f' \circ f = (\mathrm{G}' \circ \mathrm{G}, \mathrm{A}, \mathrm{C})$; this is a correspondence, with source A, target C, and graph $\mathrm{G}' \circ \mathrm{G}$. Proposition 5 implies $(f' \circ f)\langle \mathrm{A} \rangle = f'\langle f \langle \mathrm{A} \rangle \rangle$, and Proposition 3 gives $(f' \circ f)^{-1} = f^{-1} \circ f'^{-1}$, provided both correspondences are composable; two lemmas are needed; the first one says $f^{-1} \circ f'^{-1}$ is defined, the other one says that it is the LHS.

```
Definition composableC r' r :=
  is_correspondence r & is_correspondence r' & source r' = target r.
Definition compose r' r :=
  corresp (source r)(target r') (compose_graph (graph r')(graph r)).
Lemma corresp_compose: forall r' r,
  is_correspondence r -> is_correspondence r' ->
  is_correspondence (compose r' r).
Lemma compose_of_sets: forall r' r x,
  image_by_fun(compose r' r) x = image_by_fun r' (image_by_fun r x).
Lemma inverse_compose_cor: forall r r',
  inverse_fun (compose r' r) =  compose (inverse_fun r)(inverse_fun r').
```

Denote by Δ_A the diagonal of A and by I_A the identity correspondence defined by $(\Delta_\mathrm{A}, \mathrm{A}, \mathrm{A})$.

```
Definition identity_fun x := corresp x x (diagonal x).

Lemma correspondence_identity: forall x,
  is_correspondence (identity_fun x).
```

If f is a correspondence between A and B then $f \circ \mathrm{I}_\mathrm{A}$ and $\mathrm{I}_\mathrm{B} \circ f$ are equal to f. In particular $\mathrm{I}_\mathrm{A} \circ \mathrm{I}_\mathrm{A} = \mathrm{I}_\mathrm{A}$.

```
Lemma compose_identity_left: forall m,
   is_correspondence m -> compose (identity_fun (target m)) m = m.
Lemma compose_identity_right: forall m,
  is_correspondence m -> compose m (identity_fun (source m)) = m.
Lemma compose_identity_identity: forall x,
  compose (identity_fun x) (identity_fun x) =  (identity_fun x).
Lemma identity_self_inverse: forall x,
  inverse_fun (identity_fun x) =  (identity_fun x).
```

4.4 Functions

We say that r is *functional* if each x is related to at most one y. We show that this definition is equivalent to the one given in Section 3.1, that says that if z and z' are in r, then $\mathrm{pr}_1 z = \mathrm{pr}_1 z'$ implies $z = z'$. Remember that $\mathcal{V}_r x$ denotes the object v (if it exists) such that x is related to v.

```
Definition functional_graph r :=
  forall x y y', related r x y -> related r x y' ->  y=y'.

Lemma is_functional: forall r,
  (is_graph r & functional_graph r) = (fgraph r).     (* 12 *)
```

A *function* is a correspondence $f = (\mathrm{G},\mathrm{A},\mathrm{B})$ with a functional graph G, where A is the domain of G. This means that every x in A is related to unique y. This is denoted in Bourbaki by $f(x)$ or $\mathrm{G}(x)$. Here we use either $\mathcal{V}_{\mathrm{G}}x$ or $\mathcal{W}_f x$. Note: since *source* is only defined for correspondences, by type inference the expression *is_function f* implies that f is a correspondence. Note: Bourbaki says [2, p. 82] "we shall often use the word 'function' in place of 'functional graph' ".

```
Definition is_function f :=
    is_correspondence f & fgraph (graph f) & source f = domain (graph f).

Lemma fgraph_function: forall f ,
  is_function f -> fgraph (graph f).
Lemma is_graph_function: forall f,
  is_function f -> is_graph (graph f).
Lemma is_function_pr: forall s t g,
  fgraph g -> sub (range g) t -> s = domain g ->
  is_function (corresp s t g).
Lemma domain_graph: forall f, is_function f -> domain (graph f) = source f.
Lemma image_by_fun_source: forall f, is_function f ->
  image_by_fun f (source f) =  range (graph f).
Lemma related_inc_source: forall f x y,
  is_function f -> related (graph f) x y -> inc x (source f).

Lemma is_function_functional: forall f, is_correspondence f ->
  is_function f = (forall x, inc x (source f) ->
    exists_unique (fun y => related (graph f) x y)).
```

All properties of $\mathcal{V}$ give a corresponding one for $\mathcal{W}$. All lemmas listed here are trivial. Let $f = (\mathrm{G},\mathrm{A},\mathrm{B})$ be a function. If $x \in \mathrm{A}$ then $(x,\mathcal{W}_f x) \in \mathrm{G}$, $\mathcal{W}_f x \in \mathrm{range}(\mathrm{G})$ and $\mathcal{W}_f x \in \mathrm{B}$. If $y \in \mathrm{range}(\mathrm{G})$, there exists x such that $y = \mathcal{W}_f x$. If $z \in \mathrm{G}$ then $z = (\mathrm{pr}_1 z, \mathcal{W}_f \mathrm{pr}_1 z)$, $\mathrm{pr}_2 z = \mathcal{W}_f \mathrm{pr}_1 z$, and $\mathrm{pr}_1 z \in \mathrm{A}$. If $(x,y) \in \mathrm{G}$ then $y = \mathcal{W}_f x$, $x \in \mathrm{A}$ and $y \in \mathrm{B}$. Finally, if $\mathrm{X} \subset \mathrm{A}$ then $y \in f\langle \mathrm{X} \rangle$ if and only if there is $x \in \mathrm{X}$ such that $y = \mathcal{W}_f x$.

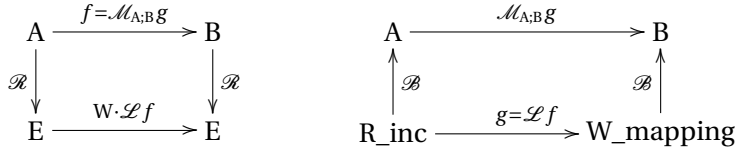
```
Definition W x f := V x (graph f).

Lemma defined_lem: forall f x,
  is_function f-> inc x (source f) -> inc (J x (W x f)) (graph f).
Lemma inc_W_range_graph:forall f x, is_function f -> inc x (source f)
  -> inc (W x f) (range (graph f)).
Lemma inc_W_target: forall f x, is_function f -> inc x (source f)
  -> inc (W x f) (target f).
Lemma range_inc_rw: forall f y, is_function f ->
  inc y (range (graph f)) = exists x:E, (inc x (source f) & y = W x f).
Lemma in_graph_W: forall f x,
  is_function f -> inc x (graph f) -> x = (J (P x) (W (P x) f)).
Lemma pr2_W: forall f x, is_function f ->
  inc x (graph f) -> Q x = W (P x) f.
Lemma inc_pr1graph_source1:forall f x, is_function f ->
```

```
  inc x (graph f) ->  inc (P x) (source f).
Lemma W_pr: forall f x y, is_function f ->
  inc (J x y) (graph f) -> W x f = y.
Lemma inc_pr1graph_source: forall f x y, is_function f ->
  inc (J x y) (graph f) ->  inc x (source f).
Lemma inc_pr2graph_target: forall f x y, is_function f ->
  inc (J x y) (graph f) ->  inc y (target f).
Lemma W_image: forall f x y, is_function f -> sub x (source f) ->
  (inc y (image_by_fun f x) = exists u, inc u x & W u f = y).
Lemma image_of_fun_pr: forall f, image_of_fun f = image_by_fun f (source f).
Lemma sub_image_target:
  forall f, is_function f -> sub (image_of_fun f) (target f).
Lemma sub_image_target1: forall g x, is_function g ->
  sub (image_by_fun g x) (target g).
```

The first lemma says $f\langle\{x\}\rangle = \{f(x)\}$. Remember that the LHS is the set of all y related to x by the function; we claim that there is exactly one such element, and is chosen by the W function. Two functions having same source, same target and same evaluation function are the same. We have $f^{-1}\langle \mathrm{B} \setminus \mathrm{X}\rangle = \mathrm{A} \setminus f^{-1}\langle \mathrm{X}\rangle$ if it is a function from A to B.

```
Lemma image_singleton: forall f x,
  is_function f -> inc x (source f) ->
  image_by_fun f (singleton x) = singleton (W x f).

Lemma funct_extensionality: forall f g,
  is_function f -> is_function g -> source f = source g ->
  target f = target g -> (forall x, inc x (source f) -> W x f = W x g)
  -> f = g.

Lemma inv_image_complement: forall g x,
  is_function g ->
  inv_image_by_fun g (complement (target g) x) =
  complement (source g) (inv_image_by_fun g x).
```

¶ Let h be a mapping (for instance $x \mapsto x+1$) and A a set (for instance the set of odd integers). We can associate a graph, namely $\mathscr{L}_{\mathrm{A}}h$. If B is another set, we can consider the function $\mathscr{L}_{\mathrm{A;B}}h$ from A to B whose graph is $\mathscr{L}_{\mathrm{A}}h$, provided that $x \in \mathrm{A}$ implies $h(x) \in \mathrm{B}$ (in the example, B must contain the even integers); this condition will be denoted by *transf_axiom* (see Section 4.6). Assume now that f maps type A into type B, its composition h with $\mathscr{R}$ is a mapping that satisfies: $x \in \mathrm{A}$ implies $h(x) \in \mathrm{B}$. The quantity $\mathscr{L}_{\mathrm{A;B}}h$ will be denoted by $\mathscr{L}f$. In the Coq source, this is *acreate*. We shall see in a moment that f can be obtained from $g = \mathscr{L}f$ by the formula $f = \mathscr{M}_{\mathrm{A;B}}g$. Lemma *W_acreate* says that the following diagram (left part) commutes.



(a/b create)

```
Lemma prop_acreate: forall (A B:Set) (f:A->B) x,
  inc x (graph (acreate f)) =  exists u:A, J(Ro u)(Ro (f u)) = x.
Lemma axioms_acreate : forall (A B:Set) (f:A->B),  fgraph(graph (acreate f)).
Lemma domain_IM: forall (A:Set) (f:A->Set),
   A = domain (IM (fun y : A => J (Ro y) (f y))).
```

```
Lemma function_acreate : forall (A B:Set) (f:A->B),  is_function(acreate f).
Lemma W_acreate : forall (A B:Set) (f:A->B) (x:A),
  W (Ro x) (acreate f) = Ro (f x).
```

Given a function g, with source A and target B, we can use the inverse function $\mathscr{B}$ of $\mathscr{R}$ to get a map f from type A to type B. We shall denote it by $\mathscr{M}g$ or $\mathscr{M}_{\mathrm{A;B}}g$. We have $\mathscr{L}f = g$. The notation $\mathscr{M}g$ is a shorthand for $\mathscr{M}_{\mathrm{source}(g);\mathrm{target}(g)}g$. If A = source($g$) and B = target($g$) but if equality is not identity then $\mathscr{M}g$ and $\mathscr{M}_{\mathrm{A;B}}g$ are objects of different type, and are not equal in Coq. In particular, if h is a mappping of type $\mathrm{A} \to \mathrm{B}$, and if $g = \mathscr{L}h$, then $\mathscr{M}g$ is a function $\mathrm{A}' \to \mathrm{B}'$, where A' is source(g) and not A, so that $\mathscr{M}\mathscr{L}h$ is not equal to h.

We create here $\mathscr{M}f$. The expression *R_inc x* is a proof of $x \in \mathrm{source}(f)$. The expression *inc_W_target* shows $w \in$ B, where B is the target of f and w the value of f. Evaluating $\mathscr{B}$ yields an object of type B, whose evaluation $\mathscr{R}$ is w. This is summarized by the first lemma. The second one says $\mathscr{L}\mathscr{M}f = f$. Remember that in order to use $\mathscr{M}f$ one needs a proof H that f is a function, and f is implicit, since it can be deduced from H.

```
Definition bcreate1 f (H:is_function f) :=
  fun x:source f => Bo (inc_W_target H (R_inc x)).

Lemma prop_bcreate1: forall f (H:is_function f) (x:source f),
  Ro(bcreate1 H x) = W (Ro x) f.
Lemma bcreate_inv1: forall f (H:is_function f),
  acreate (bcreate1 H) = f.
```

We create here $\mathscr{M}_{a;b}g$. It depends on three assumptions, g is a function, a is the source and b is the target. See diagram (a/b create) above, right part. If $x : a$, and $y = \mathscr{R}x$, the assertion *R_inc x* says $y \in a$, and applying $\mathscr{B}$ to the assertion gives y. Let $w = \mathscr{W}_g x$. The *W_mapping* lemma says (because of our three assumptions) that $w \in b$. If we apply $\mathscr{B}$, we get some element of type b, which is $\mathscr{M}_{a;b}g(x)$.

We have $\mathscr{L}\mathscr{M}_{\mathrm{A;B}}g = g$ and $\mathscr{M}_{\mathrm{A;B}}\mathscr{L}f = f$. Note: the proof of the lemma is very short. It uses the *arrow_extensionality* axiom, so that we show in fact $f'(a) = f(a)$ for all a of type A. It uses the *R_inj* axiom, so that we prove $\mathscr{R}f'(a) = \mathscr{R}f(a)$. Using *prop_back2* the lhs is the value on $\mathscr{R}a$ of $\mathscr{L}f$, which is the rhs thanks to *W_acreate*. The last lemma says that $\mathscr{M}f = \mathscr{M}_{\mathrm{A;B}}f$ for some obvious A and B.

```
Lemma W_mapping: forall f A B (Ha:source f =A)(Hb:target f =B) x,
  is_function f -> inc x A -> inc (W x f) B.

Definition bcreate f A B
  (H:is_function f)(Ha:source f =A)(Hb:target f =B):=
  fun x:A => Bo (W_mapping Ha Hb H (R_inc x)).

Lemma prop_bcreate2: forall f A B
  (H:is_function f) (Ha:source f =A)(Hb:target f=B)(x:A),
  Ro(bcreate H Ha Hb x) = W (Ro x) f.

Lemma bcreate_inv2: forall f A B
  (H:is_function f) (Ha:source f=A)(Hb:target f=B),
  acreate (bcreate H Ha Hb) = f.

Lemma bcreate_inv3: forall (A B:Set) (f:A->B),
  (bcreate (function_acreate f) (refl_equal A) (refl_equal B)) = f.
```

```
Lemma bcreate_eq: forall f (H:is_function f),
  bcreate1 H = bcreate H (refl_equal (source f)) (refl_equal (target f)).
```

Let's consider some examples of functions. The empty function is the only function from the empty set to itself; its graph is empty. Note that if the graph is empty, so is the source. [2]

```
Definition empty_functionC : emptyset -> emptyset := fun x => x.
Definition empty_function:= acreate empty_functionC.

Lemma function_empty_function: is_function empty_function.
Lemma graph_empty_function: graph empty_function = emptyset.
Lemma special_empty_function: forall f, is_function f ->
  graph f = emptyset -> source f = emptyset.
Lemma empty_function_prop:
  bcreate function_empty_function (refl_equal (source empty_function))
  (refl_equal (target empty_function))
  = empty_functionC.
```

¶ We have already met the identity function. The properties shown here are trivial.

```
Lemma function_identity: forall x,
  is_function (identity_fun x).
Lemma W_identity: forall x y,
  inc y x -> W y (identity_fun x) = y.
```

We define *identityC a* to be the identity on *a* as a Coq function. By default, the argument *a* is implicit; we make it explicit.

```
Definition identityC (a:Set): a->a := fun x => x.
Implicit Arguments identityC [].

Lemma w_identity: forall a x,  identityC a x = x.
Lemma identity_prop: forall a, acreate (identityC a) = identity_fun a.
Lemma identity_prop2: forall a,
  bcreate (function_identity a) (refl_equal (source (identity_fun a)))
  (refl_equal (target (identity_fun a))) =
  identityC a.
```

One of theorems in Chapter 2 (part II of this report) will be of the form: if P is true, there exists a unique function f such that Q. This function is denoted by $\tau_f(Q)$ in Bourbaki. We cannot apply *choose* in Coq, since it gives a set, and not a function. Thus we define *choosef*, a variant of $\tau_f(Q)$ that produces the identity function of the empty set if no function f satisfies Q.

```
Definition choosef (p:correspondenceC -> Prop) :=
  chooseT (fun u=> (ex p -> p u)  &  ~(ex p) -> u = identity_fun emptyset)
  (nonemptyT_intro (corresp emptyset emptyset emptyset)).

Lemma choosef_pr : forall p, (ex p) -> p (choosef p).
```

[2] Some trivial results, such as: "the source of the empty function is the empty set", have been removed in V3 since they are consequence of the *simpl* tactic. These lemmas can be found in the last chapter

¶ If $a \in x$ and $b \in x$ imply $a = b$, we say that x is a small set. It is either empty or has a single element. We say that a function is *constant* if it takes at most one value; in other words that the range is a small set (if the source is not empty, the range is nonempty).

The constant function $C_{xy}a$ maps b of type x to a of type y; for this reason, arguments x and y are implicit. The Bourbaki function $\Gamma = (x \times \{a\}, x, y)$ needs the assumption $a \in y$. Hence a and y are implicit. We make all parameters explicit.

```
Definition small_set x :=
  forall u v, inc u x -> inc v x -> u = v.

Definition is_constant_function f :=
  (is_function f ) &
  (forall x x', inc x (source f) -> inc x' (source f) -> W x f = W x' f).

Definition constant_functionC x y (a:y)  :=  fun _:x => a.
Implicit Arguments constant_functionC [].
Definition constant_function (x y a:Set) (H:inc a y) :=
  acreate (constant_functionC x y (Bo H)).
Implicit Arguments constant_function [].
```

These are the basic properties of constant functions.

```
Lemma source_constant: forall x y a (H:inc a y),
  source (constant_function x y a H) = x.
Lemma target_constant: forall x y a  (H:inc a y),
  target (constant_function x y a H) = y.
Lemma graph_constant: forall x y a  (H:inc a y),
  graph (constant_function x y a H) = product x (singleton a).
Lemma function_constant_fun: forall x y a (H: inc a y),
  is_function(constant_function x y a H).
Lemma W_constant: forall x y a (H:inc a y) b,
  inc b x ->  W b (constant_function x y a H) = a.
Lemma w_constant_functionC: forall  x y (a:y)  (z:x),
  constant_functionC x y a z = a.
```

We give now the link between constant functions, and the property of being constant. Every constant function is of the form $C_{xy}a$ for some a if x is not empty. In the case of a Bourbaki function, instead of saying "there exists $a \in y$" we say "there exists a of type y" from which we deduce an element and the proof that it is in y.

```
Lemma constant_function_pr: forall f,
  is_function  f -> (is_constant_function f =
    small_set (range (graph f))).
Lemma constant_constant_fun: forall x y a (H: inc a y),
 is_constant_function(constant_function x y a H).
Lemma constant_fun_constantC: forall  x y a,
   is_constant_functionC (constant_functionC x y a).
Lemma constant_function_prop2: forall (x y:Set) (a:y),
  bcreate (function_constant_fun x (R_inc a)) (source_constant x (R_inc a))
  (target_constant x (R_inc a)) = constant_functionC x y a.
Lemma constant_fun_prC: forall x y (f:x->y)(b:x), is_constant_functionC f ->
  exists a:y, f= constant_functionC x y a.
Lemma nonempty_target: forall f,
  nonempty(graph f) -> inc (rep (target f)) (target f).
```

```
Lemma constant_fun_pr: forall f (H:nonempty(graph f)),
  is_constant_function f ->
  exists a: target f,
  f= constant_function (source f) (target f) (Ro a) (R_inc a).
```

4.5 Restrictions and extensions of functions

The *restrictionrestriction* of a function f to a set x can be defined in different ways, for instance as the composition with the inclusion map from x in the source of f. This is the definition we shall use for Coq functions. In Bourbaki, composition is defined for correspondences, and the case of functions is studied later, in Section 4.7.

We define here the composition of two Coq functions; associativity is trivial, it suffices to unfold the definitions. Identity is a unit; this relies on the fact that f is equal to the function that maps u to $f(u)$.

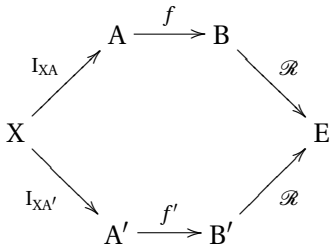
```
Definition composeC a b c (g:b->c) (f: a->b):= fun x:a => g (f x).
Lemma compositionC_associative: forall a b c d (f: c->d)(g:b->c)(h:a->b),
  composeC (composeC f g) h = composeC f (composeC g h).
Lemma compose_id_leftC: forall (a b:Set) (f:a->b),
  composeC (identityC b) f = f.
Lemma compose_id_rightC: forall (a b:Set) (f:a->b),
  composeC f (identityC a) = f.
```

$$\begin{array}{ccc} x & \xrightarrow{\mathrm{I}_{x;y}} & y \\ \Big\downarrow{\scriptstyle \mathscr{R}} & & \Big\uparrow{\scriptstyle \mathscr{B}} \\ \mathrm{R\_inc} & \xrightarrow{\subset} & \mathrm{H\_sub} \end{array} \qquad \text{(inclusion)}$$

We now define the inclusion I_{xy}. See diagram (inclusion) which is an instance of (a/b create). If x and y are two sets, H is the assumption $x \subset y$, if u is of type x, then *R_inc u* says that $\mathscr{R}u \in x$. Applying H gives $\mathscr{R}u \in y$, denoted by *H_sub* on the diagram, and using $\mathscr{B}$ yields an object of type y. The important property is $\mathscr{R}a = \mathscr{R}(\mathrm{I}_{xy}(a))$. From the injectivity of $\mathscr{R}$ we deduce $\mathrm{I}_{xx} = \mathrm{I}_x$ and $\mathrm{I}_{yz} \circ \mathrm{I}_{xy} = \mathrm{I}_{xz}$ (where *sub_refl* says $x \subset x$ and *sub_trans* expresses the transitivity of inclusion, in other words, it says that if $\mathrm{I}_{yz} \circ \mathrm{I}_{xy}$ is defined so is I_{xz}).

```
Definition inclusionC x y (H: sub x y):=
  fun u:x => Bo (H (Ro u) (R_inc u)).

Lemma inclusionC_pr: forall x y (H: sub x y) (a:x),
  Ro(inclusionC H a) = Ro a.
Lemma inclusionC_identity: forall x,
  inclusionC (sub_refl (x:=x)) = identityC x.
Lemma inclusionC_compose: forall x y z (Ha:sub x y)(Hb: sub y z),
  composeC (inclusionC Hb)(inclusionC Ha) = inclusionC (sub_trans Ha Hb).
```

We say that two functions agree on a set x if this set is a subset of the sources, and if the functions take the same value on x. Consider two functions $(\mathrm{G},\mathrm{A},\mathrm{B})$ and $(\mathrm{G}',\mathrm{A}',\mathrm{B}')$. If $\mathrm{A} = \mathrm{A}'$ and $\mathrm{G} = \mathrm{G}'$, the functions agree on A. Conversely, the property "$\mathrm{A} \subset \mathrm{A}'$ and the functions agree on A" is the same as $\mathrm{G} \subset \mathrm{G}'$. Thus if $\mathrm{A} = \mathrm{A}'$ we have $\mathrm{G} = \mathrm{G}'$. If moreover $\mathrm{B} = \mathrm{B}'$, the functions are the same.



(restriction/agree)

In the case of Coq functions, $f : A \to B$ and $f' : A' \to B'$ agree on X if the diagram (restriction/agree) commutes.

```
Definition agrees_on x f f' :
  (sub x (source f)) & (sub x (source f')) &
  (forall a, inc a x -> W a f = W a f').

Definition restrictionC (x a b:Set) (f:a->b)(H: sub x a) :=
  composeC f (inclusionC H).
Definition agreeC (x a a' b b':Set) (f:a->b) (f':a'-> b')
  (Ha: sub x a)(Hb: sub x a') :=
  forall u:x, Ro(restrictionC  f Ha u) = Ro(restrictionC f' Hb u).

Lemma same_graph_agrees: forall f f',
  is_function f -> is_function f' -> graph f = graph f' ->
  agrees_on (source f) f f'.

Lemma function_extens: forall f f',
  is_function f -> is_function f' ->
  (f =f')= ((source f =source f') & (target f= target f') &
    (agrees_on (source f) f f')).

Lemma sub_function: forall f g,
  is_function f -> is_function g ->
  (sub (graph f) (graph g)) = ((sub (source f) (source g))
    & (agrees_on (source f) f g)).
```

¶ We consider here the restriction of a function to a subset of its domain. We need some lemmas in order to define it. We have two possibilities: the target of the new function can be either the target of the old function, or the range.

```
Definition restriction_function f x :=
  corresp x (target f) (restr (graph f) x).
Definition restriction1 f x :=
  corresp x (image_by_fun f x) (restr (graph f) x).

Lemma restriction_graph_is_graph: forall f x,
  is_function f-> fgraph (restr (graph f) x).
Lemma domain_restriction_graph: forall f x,
  is_function f-> sub x (source f) ->
  domain (restr (graph f) x) = x.
Lemma range_restriction_graph: forall f x,
  is_function f-> sub x (source f) ->
  sub (range (restr (graph f) x)) (target f).
Lemma function_restriction: forall f x,
```

```
  is_function f -> sub x (source f) ->
  is_function (restriction_function f x).

Lemma function_restriction1: forall f x,
  is_function f -> sub x (source f) ->
  is_function (restriction1 f x).
Lemma W_restriction: forall f x a,
  is_function f -> sub x (source f) ->
  inc a x ->  W a (restriction_function f x) = W a f.
Lemma W_restriction1: forall f x a,
  is_function f -> sub x (source f) ->
  inc a x ->  W a (restriction1 f x) = W a f.
```

We say that $g = (G, C, D)$ *extends* $f = (F, A, B)$ if $F \subset G$ and $B \subset D$. This implies $A \subset C$. Both functions agree on A. In the case of Coq functions, there is no notion of graph, hence: for every f and g such that the source of f is a subset of the source of g, we say that g extends f if the target of f is a subset of the target of g, and if the functions agree on the source of f.

```
Definition extends g f :=
  (is_function f) & (is_function g) & (sub (graph f) (graph g))
  & (sub (target f)(target g)).
Definition extendsC (a b a' b':Set) (g:a'->b')(f:a->b)(H: sub a a') :=
  sub b b' & agreeC g f H (sub_refl (x:=a)).

Lemma source_extends: forall f g,
  extends g f -> sub (source f) (source g).

Lemma W_extends: forall f g x,
  extends g f -> inc x (source f) -> W x f = W x g.

Lemma extendsC_pr : forall  (a b a' b':Set) (g:a'->b')(f:a->b)(H: sub a a'),
  extendsC g f H -> forall x:a, Ro (f x) = Ro(g (inclusionC H x)).
```

If f is a function, X a subset of its source, then f extends its restriction to X. If f and g are two functions with the same target, that agree on X, their restrictions to X are equal. The same is true for Coq functions. Bourbaki notes that the graph of the restriction is the intersection with the product of X and the target (but he cannot prove this statement, since intersection is not yet defined).

```
Lemma function_extends_rest: forall f x,
  is_function f -> sub x (source f) ->
Lemma function_extends_restC: forall (x a b:Set) (f:a->b)(H:sub x a),
  extendsC f (restrictionC f H) H.
Lemma agrees_same1: forall f g x, agrees_on x f g -> sub x (source f).
Lemma agrees_same2: forall f g x, agrees_on x f g -> sub x (source g).
Lemma agrees_same_restriction: forall f g x,
  is_function f -> is_function g -> agrees_on x f g ->
  target f = target g ->
  restriction_function f x = restriction_function g x.
Lemma agrees_same_restrictionC: forall (a a' b x:Set) (f:a->b)(g:a'->b)
  (Ha: sub x a)(Hb: sub x a'),
  agreeC f g Ha Hb -> restrictionC f Ha = restrictionC g Hb.
Lemma restriction_graph1: forall f x,
  is_function f -> sub x (source f) ->
  graph (restriction_function f x) =
```

```
  intersection2 (graph f)(product x (target f)).
Lemma restriction_create: forall f x,
  is_function f -> sub x (source f) ->
  restriction_function f x = create x (target f)
  (intersection2 (graph f)(product x (target f))).
```

¶ Given a function $f = (G, A, B)$ and two sets X and Y, we can consider $(G \cap (X \times B), X, Y)$. This is a function if $X \subset A$, $Y \subset B$ and the image of X by f is a subset of Y (a name is given to this condition). The function agrees with f on X. If f is the extension of some function g, then g is the restriction of f to its source and target.

```
Definition restriction2 f x y :=
  corresp x y (intersection2 (graph f) (product x (target f))).

Definition restriction2_axioms f x y :=
  is_function f &
  sub x (source f) & sub y (target f) & sub (image_by_fun f x) y.

Lemma graph_restriction2: forall f x y,
  sub (graph (restriction2 f x y)) (graph f).
Lemma inc_graph_restriction2: forall f x y a b,
  (inc (J a b)  (graph (restriction2 f x y))) =
  (inc (J a b) (graph f) & inc a x & inc b (target f)).
Lemma function_restriction2: forall  f x y,
  restriction2_axioms f x y ->
  is_function (restriction2 f x y).    (* 19 *)
Lemma W_restriction2: forall f x y a,
  restriction2_axioms f x y ->
  inc a x -> W a  (restriction2 f x y) = W a f.
Lemma function_rest_of_prolongation: forall f g,
  extends g f -> f = restriction2 g (source f) (target f).
```

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ {\scriptstyle \mathrm{I}_{ac}}\downarrow & & \downarrow{\scriptstyle \mathrm{I}_{bd}} \\ c & \xrightarrow[\mathrm{R}_{cd}f]{} & d \end{array} \qquad \text{(restriction2C)}$$

In the case of Coq functions, we start with a function $f : a \to b$, with the assumptions $c \subset a$ and $d \subset b$. The restriction $\mathrm{R}_{cd}f$ is the one that makes diagram (restriction2C) commute. In order for it to exist, each y in the image of the lhs must be convertible to type d, i.e. $\mathscr{R}y \in d$.

```
Definition restriction2C (a a' b b':Set) (f:a->b)(Ha:sub a' a)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b') :=
  fun u=> Bo (H u).
Lemma restriction2C_pr: forall (a a' b b':Set) (f:a->b)(Ha:sub a' a)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b') (x:a'),
  Ro (restriction2C f Ha H x) = W (Ro x) (acreate f).
Lemma restriciton2C_pr1: forall (a a' b b':Set)  (f:a->b)
  (Ha:sub a' a)(Hb:sub b' b)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b'),
  composeC f (inclusionC Ha) = composeC (inclusionC Hb) (restriction2C f Ha H).
```

4.6 Definition of a function by means of a term

In Bourbaki [2, p. 83], Criterion C54 says that if $\boldsymbol{A}$ and $\boldsymbol{T}$ are two terms, $\boldsymbol{x}$ and $\boldsymbol{y}$ are two distinct letters, $\boldsymbol{x}$ is not in $\boldsymbol{A}$, $\boldsymbol{y}$ is neither in $\boldsymbol{T}$ nor in $\boldsymbol{A}$, then the relation $\boldsymbol{x} \in \boldsymbol{A}$ and $\boldsymbol{y} = \boldsymbol{T}$ admits a graph $\boldsymbol{F}$, which is functional, and $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{T}$. If $\boldsymbol{C}$ is a set which contains the set $\boldsymbol{B}$ of objects of the form $\boldsymbol{T}$ for $\boldsymbol{x} \in \boldsymbol{A}$ (where $\boldsymbol{y}$ does not appear in $\boldsymbol{C}$), the function $(\boldsymbol{F}, \boldsymbol{A}, \boldsymbol{C})$ is also denoted by the notation $\boldsymbol{x} \rightarrow \boldsymbol{T}$ $(\boldsymbol{x} \in \boldsymbol{A}, \boldsymbol{T} \in \boldsymbol{C})$, where the terms in parentheses may be omitted. It can also be written as $(\boldsymbol{T})_{\boldsymbol{x} \in \boldsymbol{A}}$. In what follows, we shall use $x \mapsto \mathrm{T}$ to denote the function that associates T to x, and $x \rightarrow \mathrm{T}$ to mean a function from the set (or type) x to the set (or type) T.

The non-trivial point is the existence of the set $\boldsymbol{B}$, since $\boldsymbol{F}$ is then a subset of $\boldsymbol{A} \times \boldsymbol{B}$. The range of $\boldsymbol{F}$ is $\boldsymbol{B}$, so that $(\boldsymbol{F}, \boldsymbol{A}, \boldsymbol{C})$ is a function when $\boldsymbol{B} \subset \boldsymbol{C}$. In these definitions, $\boldsymbol{y}$ is just an auxiliary letter (because it neither appears in $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{T}$ nor $\boldsymbol{F}$). On the other hand, $\boldsymbol{x}$ may appear in $\boldsymbol{T}$, it does not appear in $\boldsymbol{A}$, $\boldsymbol{B}$, nor $\boldsymbol{F}$.

If we have an object $f : \mathrm{E} \rightarrow \mathrm{E}$, and consider $\mathrm{T} = f(x)$ then $\mathrm{F} = \mathscr{L}_{\mathrm{A}} f$. The second claim of the criterion, namely $\mathrm{F}(x) = \mathrm{T}$, is just $\mathscr{V}(x, \mathscr{L}_{\mathrm{A}} f) = f(x)$ (see Section 3.1). The function (F, A, C) will be denoted by $BL\, f\, A\, C$, or $\mathscr{L}_{\mathrm{A;C}} f$. The following lemmas are obvious from the definitions of $\mathscr{L}_{\mathrm{A}}$ and $\mathscr{L}_{\mathrm{A;C}}$.

```
Definition fun_function f a b :=
  corresp a b (L a f).

Notation BL := fun_function.
Lemma af_graph1: forall f a b c,
  inc c (graph (BL f a b)) -> c = J (P c) (f (P c)).
Lemma af_graph2: forall f a b c,
  inc c a -> inc (J c (f c)) (graph (BL f a b)).
Lemma af_graph3: forall f a b c,
  inc c (graph (BL f a b)) -> inc (P c) a.
Lemma af_graph4: forall f a b c,
  inc c (graph (BL f a b)) -> f (P c) = (Q c).
```

The expression $\mathscr{L}_{\mathrm{A;B}} f$ is a function if f maps A into B. If $x \in \mathrm{A}$, the value at x is $f(x)$. By extensionality, if f is a function with source A, target B, and evaluation function $\mathscr{W}_f$, then $\mathscr{L}_{\mathrm{A;B}} \mathscr{W}_f = f$.

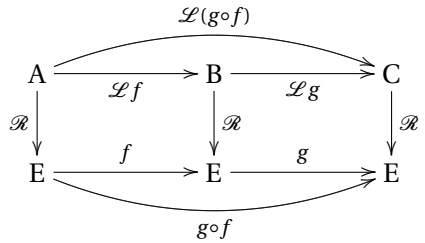
```
Definition transf_axioms f a b :=
  forall c, inc c a -> inc (f c) b.

Lemma af_function: forall f a b,
  transf_axioms f a b -> is_function (BL f a b).

Lemma W_af_function: forall f a b c,
  transf_axioms f a b ->
  inc c a-> W c (BL f a b) = f c.
Lemma af_self: forall f,
  is_function f -> BL (fun z => W z f)(source f)(target f) = f.
```

We consider here an example of a function defined by a term, the first and second projection, denoted pr_1 and pr_2, on the range and target.

```
Definition first_proj g := BL P g (domain g).
Definition second_proj g := BL Q g (range g).
```

```
Lemma W_first_proj: forall g x,
  is_graph g -> inc x g -> W x (first_proj g) = P x.
Lemma W_second_proj: forall g x,
  is_graph g-> inc x g -> W x (second_proj g) = Q x.
Lemma function_first_proj:forall g,  is_graph g -> is_function (first_proj g).
Lemma function_second_proj:forall g, is_graph g -> is_function (second_proj g).
```

4.7 Composition of two functions. Inverse function



(composition)

We say that $f = (\mathrm{F},\mathrm{A},\mathrm{B})$ and $g = (\mathrm{G},\mathrm{B},\mathrm{C})$ are *composable* if they are functions and if they are composable as correspondences. Their graphs are *fcomposable*. Proposition 6 [2, p. 84] says that the composition is a function. The evaluation function is the composition of the evaluation functions. We have $\mathscr{L}(g\circ f) = (\mathscr{L}g)\circ(\mathscr{L}f)$. In other words, the two definitions of composition (for Bourbaki and Coq functions) are really the same.

```
Definition composable g f :=
  is_function g & is_function f &  source g = target f.

Lemma composable_pr: forall f g, composable g f ->
  fcomposable (graph g) (graph f).
Lemma composable_pr1: forall f g, composable g f ->
  graph (compose g f) = fcompose (graph g) (graph f).
Lemma domain_compose:forall g f,
  composable g f->
  domain (graph(compose g f)) = domain (graph f).

Theorem is_function_compose: forall g f, composable g f
  -> is_function (compose g f).

Lemma W_compose: forall g f x,  composable g f ->
  inc x (source f) -> W x (compose g f) = W (W x f) g.

Lemma composable_acreate:forall (a b c:Set) (f: a-> b)(g: b->c),
  composable (acreate g) (acreate f).
Lemma compose_acreate:forall (a b c:Set) (g: b->c)(f: a-> b),
  compose (acreate g) (acreate f) = acreate(composeC g f).
```

Composition is associative, and identity is a unit.

```
Lemma compose_assoc: forall f g h,
  composable f g -> composable g h ->
  compose (compose f g) h = compose f (compose g h).
Lemma compose_id_left: forall m,
```

```
  is_function m-> compose (identity_fun (target m)) m = m.
Lemma compose_id_right: forall m,
  is_function m-> compose m  (identity_fun (source m)) = m.
```

We say that f is *injective* if it is a function such that $f(x) = f(y)$ implies $x = y$. We say that f is *surjective* if the range of its graph is the target. The phrase "f is a mapping of A onto B" is sometimes used by Bourbaki as a shorthand of "f is surjective, its source is A, and its target is B". We say that f is *bijective* if it satisfies both properties. We list here some trivial properties.

```
Definition injective f:=
  is_function f &
  (forall (x y), inc x (source f) -> inc y (source f) ->
    W x f = W y f -> x = y).

Definition surjective f :=
  is_function f & image_of_fun f = target f.

Definition bijective f :=
  injective f & surjective f.

Definition equipotent x y :=
  exists z, bijective z & source z = x & target z = y.


Lemma inj_is_function: forall f, injective f -> is_function f.
Lemma surj_is_function: forall f, surjective f-> is_function f.
Lemma bij_is_function: forall f, bijective f-> is_function f.


Lemma injective_pr: forall f x x' y,
  injective f -> related (graph f) x y -> related (graph f) x' y -> x = x'.
Lemma injective_pr3: forall f x x' y,
  injective f -> inc (J x y) (graph f) -> inc (J x' y) (graph f) -> x = x'.
Lemma injective_pr_bis: forall f,
  is_function f -> (forall x x' y,
  related (graph f) x y -> related (graph f) x' y -> x = x') -> injective f.
Lemma surjective_pr: forall f y,
  surjective f -> inc y (target f) ->
  exists x, inc x (source f) & related (graph f) x y .
Lemma surjective_pr2: forall f y,
  surjective f -> inc y (target f) -> exists x, inc x (source f) & W x f = y.
Lemma surjective_pr3: forall f,
  surjective f -> range (graph f) = target f.
Lemma surjective_pr4: forall f,
  is_function f-> range (graph f) = target f -> surjective f.
Lemma surjective_pr5: forall f,
  is_function f -> (forall y, inc y (target f) ->
  exists x, inc x (source f) & related (graph f) x y) -> surjective f.
Lemma surjective_pr6: forall f,
  is_function f-> (forall y, inc y (target f) ->
  exists x, inc x (source f) & W x f = y) -> surjective f.
Lemma injective_af_function:  forall f a b, transf_axioms f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  injective (BL f a b).
Lemma surjective_af_function: forall f a b, transf_axioms f a b ->
  (forall y, inc y b -> exists x, inc x a & f x = y) ->
```

```
  surjective (BL f a b).
Lemma bijective_af_function: forall f a b, transf_axioms f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  (forall y, inc y b -> exists x, inc x a & f x = y) ->
  bijective (BL f a b).
Lemma bijective_pr: forall f y,
  bijective f -> inc y (target f) ->
  exists_unique (fun x=> inc x (source f) & W x f = y).
```

Let's consider the case of Coq functions. We do not need as many lemmas, because they are trivialities. Functors *acreate* and *bcreate* map bijections to bijections. We say that two sets are *equipotent* if there is a bijection between them. Which definition of bijection used is irrelevant.

```
Definition injectiveC (a b:Set) (f:a->b) := forall u v, f u = f v -> u =v.
Definition surjectiveC (a b:Set) (f:a->b) := forall u, exists v, f v = u.
Definition bijectiveC (a b:Set) (f:a->b) := injectiveC f & surjectiveC f.


Lemma bijectiveC_pr: forall (a b:Set) (f:a->b) (y:b),
  bijectiveC f ->   exists_unique (fun x:a=> y = f x).

Lemma equipotentC: forall x y, equipotent x y = exists f:x->y, bijectiveC f.

Lemma injective_bcreate: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  injective f -> injectiveC (bcreate H Ha Hb).
Lemma surjective_bcreate: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  surjective f -> surjectiveC (bcreate H Ha Hb).
Lemma bijective_bcreate: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  bijective f -> bijectiveC (bcreate H Ha Hb).

Lemma injective_bcreate1: forall f (H:is_function f),
  injective f -> injectiveC (bcreate1 H).
Lemma surjective_bcreate1: forall f (H:is_function f),
  surjective f -> surjectiveC (bcreate1 H).
Lemma bijective_bcreate1: forall f (H:is_function f),
  bijective f -> bijectiveC (bcreate1 H).

Lemma injective_acreate: forall (a b:Set) (f:a->b),
  injectiveC f -> injective (acreate f).
Lemma surjective_acreate: forall (a b:Set) (f:a->b),
  surjectiveC f -> surjective (acreate f).
Lemma bijective_acreate: forall (a b:Set) (f:a->b),
  bijectiveC f -> bijective (acreate f).
Lemma equipotentC: forall x y, equipotent x y = exists f:x->y, bijectiveC f.
```

The identity function is bijective; the restriction of a function f to X and Y is injective if f is injective; it is surjective if for instance X is the source and Y the range. It is surjective if $Y = f(X)$.

```
Lemma bijective_identity: forall x,
  bijective (identity_fun x).
Lemma bijective_identityC: forall x,
```

```
  bijectiveC (identityC x).
Lemma injective_restriction2: forall f x y,
  injective f -> restriction2_axioms f x y
  -> injective (restriction2 f x y).
Lemma surjective_restriction2: forall f x y,
  restriction2_axioms f x y ->
  source f = x -> image_of_fun f = y ->
  surjective (restriction2 f x y).

Lemma surjective_restriction1: forall f x,
  is_function f -> sub x (source f) ->
  surjective (restriction1 f x).
Lemma bijective_restriction1: forall f x,
  injective f -> sub x (source f) ->
  bijective (restriction1 f x).
```

¶ The canonical injection of A into B is the identity of B restricted to A. In other terms, if $A \subset B$ it is the function with source A, target B, whose evaluation function is $x \mapsto x$. It is injective with range A. Its Coq equivalent has been introduced page 54.

```
Definition canonical_injection a b :=
  corresp a b (diagonal a).
Lemma function_ci: forall a b, sub a b ->
  is_function  (canonical_injection a b).
Lemma W_ci: forall a b x,
  sub a b -> inc x a -> W x (canonical_injection a b) = x.
Lemma injective_ci: forall a b,
  sub a b -> injective (canonical_injection a b).
Lemma range_ci: forall a b, sub a b ->
  range (graph (canonical_injection a b)) = a.
```

¶ The diagonal application is the function from X to $X \times X$ that maps x to (x,x). It is an injection into the diagonal of X.

```
Definition diagonal_application a :=
  BL (fun x=> J x x) a (product a a).
Lemma graph_diag_app: forall a x,
  inc x  (graph (diagonal_application a)) =
  (is_pair x & inc (P x) a & Q x = J (P x) (P x)).
Lemma W_diag_app:forall a x,
  inc x a -> W x (diagonal_application a) = J x x.
Lemma function_diag_app: forall a, is_function (diagonal_application a).
Lemma injective_diag_app: forall a,  injective (diagonal_application a).
Lemma range_diag_app: forall a,
  range (graph (diagonal_application a)) = diagonal a.
```

¶ Both projections pr_1 and pr_2 are surjective by construction. The first projection on G is injective if only if G is a functional graph.

```
Lemma surjective_second_proj: forall g,
  is_graph g -> surjective (second_proj g).
Lemma surjective_first_proj: forall g ,
  is_graph g -> surjective (first_proj g).
Lemma injective_first_proj: forall g,
  is_graph g -> (injective (first_proj g) = functional_graph g).
```

¶ If G is a graph, the map $(x, y) \mapsto (y, x)$ maps G onto G^{-1} (as noted when we defined the inverse graph). From this, we get a bijective function. Bourbaki considers the following function: we fix a, and map x to (x, a). This is a bijection between X and the product $X \times \{b\}$. This could be restated as: X is equipotent to $X \times Y$ when Y is a singleton.

```
Definition inv_graph_canon g :=
   BL (fun x=> J (Q x) (P x)) g (inverse_graph g).

Lemma W_inv_graph_canon: forall g x,
  is_graph g ->inc x g -> W x (inv_graph_canon g) = J (Q x) (P x).
Lemma function_inv_graph_canon:  forall g,
  is_graph g -> is_function (inv_graph_canon g).
Lemma bijective_inv_graph_canon:  forall g,
  is_graph g -> bijective (inv_graph_canon g).   (* 15 *)

Lemma bourbaki_ex5_17: forall a b,
  bijective ( BL (fun x=> J x b) a (product a (singleton b))).    (* 14 *)
Lemma equipotent_prod_singleton:
  forall a b, equipotent a (product a (singleton b)).
Lemma restriction1_pr: forall f,
  is_function f -> restriction2 f (source f) (image_by_fun f (source f)) =
  restriction1 f (source f).
Lemma rest_to_image_surjective: forall f,
  is_function f -> surjective (restriction2 f (source f)
    (image_by_fun f (source f))).
```

Proposition 7 [2, p. 85] states that if f is a bijection, then the inverse correspondence f^{-1} is a function. It also says that if f and f^{-1} are functions then f is a bijection.

```
Theorem bijective_inv_function: forall f,
  bijective f -> is_function (inverse_fun f).
Theorem inv_function_bijective: forall f,
  is_function f -> is_function (inverse_fun f) -> bijective f.
```

In the case of a Coq function f, its inverse is defined by $f^{-1} = \mathscr{M}_{b;a}((\mathscr{L}f)^{-1})$. We need a bunch of trivial lemmas in order to use $\mathscr{M}$. This function satisfies $\mathscr{L}(f^{-1}) = (\mathscr{L}f)^{-1}$.

We have $f^{-1}(f(x)) = x$. By injectivity of $\mathscr{R}$, it suffices to show that $\mathscr{R}f^{-1}(f(x)) = \mathscr{R}x$. Using *bcreate*, it suffices to show that $\mathscr{W}_{\tilde{f}^{-1}}(\mathscr{R}f(x)) = \mathscr{R}x$. Denoting $y = \mathscr{R}x$ and $g = \tilde{f}$, we have to show that $\mathscr{W}_{g^{-1}}(\mathscr{W}_g(y)) = y$. This is a consequence of $g^{-1} \circ g = \mathrm{I}$, relation to be shown later. But since we know that g and g^{-1} are functions, we can restate this as follows: the pair $(\mathscr{W}_g(y), y)$ is in the inverse graph of g, which is the same as: the pair $(y, \mathscr{W}_g(y))$ is in the graph of g, which is true. From this we deduce $f(f^{-1}(f(x))) = f(x)$, and the surjectivity of f gives $f \circ f^{-1} = \mathrm{I}$.

```
Lemma bijective_inv_aux: forall (a b:Set) (f:a->b),
  bijectiveC f -> is_function (inverse_fun (acreate f)).
Lemma bijective_source_aux: forall (a b:Set) (f:a->b),
  source (inverse_fun (acreate f)) = b.
Lemma bijective_target_aux: forall (a b:Set) (f:a->b),
  target (inverse_fun (acreate f)) = a.

Definition inverseC a b (f:a->b)(H:bijectiveC f): b->a :=
  bcreate(bijective_inv_aux H)(bijective_source_aux f)
```

```
  (bijective_target_aux f).

Lemma inverseC_pra: forall (a b:Set) (f:a->b)(H:bijectiveC f) (x:a),
  (inverseC H) (f x) = x.
Lemma inverseC_prb: forall (a b:Set) (f:a->b)(H:bijectiveC f) (x:b),
  f ((inverseC H) x) = x.
Lemma inverseC_prc: forall (a b:Set) (f:a-> b) (H:bijectiveC f),
  inverse_fun(acreate f) = acreate(inverseC H).
Lemma bij_left_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f) ,
  composeC (inverseC H) f = identityC a.
Lemma bij_right_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f) ,
  composeC f (inverseC H) = identityC b.
```

If a function has a left and right inverse, the function is bijective, and its inverse is equal to these inverses. In fact, if $f(g(x)) = x$ for all x, then f is surjective, since every x is the image of $g(x)$. If $g'(f(y)) = y$, applying g' to $f(y) = f(y')$ gives $y = y'$, hence proves injectivity. Now, $g'(f(g(x))) = g'(x) = g(x)$, this shows that $g = g'$. We have $g'(x) = f^{-1}(x)$, since $x = f(g(x))$, and, by definition, the rhs is $g(x)$. We have already seen that the lhs is this quantity.

We deduce from this that the inverse function of a bijection is a bijection.

```
Lemma bijective_double_inverseC: forall (a b:Set) (f:a->b) g g',
  composeC  g f = identityC a -> composeC  f g' = identityC b ->
  bijectiveC f.
Lemma bijective_double_inverseC1: forall (a b:Set) (f:a->b) g g'
  (Ha: composeC  g f = identityC a)(Hb: composeC  f g' = identityC b),
  g = inverseC(bijective_double_inverseC Ha Hb)
  & g' = inverseC(bijective_double_inverseC Ha Hb).
Lemma bijective_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f),
  bijectiveC (inverseC H).
Lemma inverse_fun_involutiveC:forall (a b:Set) (f:a->b) (H: bijectiveC f),
  f = inverseC(bijective_inverseC H).
```

If f is a bijective, then f^{-1} is also a bijective. The composition in any order is the identity function. The proofs of these three lemmas are similar: let $g = \mathcal{M}_{a;b}f$; then $f = \mathcal{L}g$, $f^{-1} = \mathcal{L}(g^{-1})$, $f \circ f^{-1} = \mathcal{L}(g \circ g^{-1})$.

```
Lemma inverse_bij_is_bij:forall f,
  bijective f ->  bijective (inverse_fun f).

Lemma composable_f_inv:forall f,
  bijective f ->  composable f (inverse_fun f).
Lemma composable_inv_f: forall f,
  bijective f ->  composable (inverse_fun f) f.
Lemma bij_right_inverse:forall f,
  bijective f ->  compose f (inverse_fun f) = identity_fun (target f).
Lemma bij_left_inverse:forall f,
  bijective f ->  compose (inverse_fun f) f = identity_fun (source f).

Lemma W_inverse: forall f x y,
  bijective f -> inc x (target f) ->
  (y = W x (inverse_fun f)) -> (x = W y f).
Lemma W_inverse2: forall f x y,
  bijective f -> inc y (source f) ->
  (x = W y f)-> (y = W x (inverse_fun f)).
```

```
Lemma W_inverse3: forall f x,
  bijective f -> inc x (target f) -> inc (W x (inverse_fun f)) (source f).
```

We apply the results of Coq functions to Bourbaki functions. Note that Bourbaki shows that the inverse $h = f^{-1}$ is a bijection by noting that its inverse is f, hence is a function and Proposition 7 [2, p. 85] applies. The relation $x = \mathscr{W}_f y$ is equivalent to $y = \mathscr{W}_{f^{-1}} x$ if either x is in the target of f or y in the source.

```
Lemma bijective_inv_aux: forall a b (f:a->b),
  bijectiveC f -> is_function (inverse_fun (acreate f)).
Lemma bijective_source_aux: forall a b (f:a->b),
  source (inverse_fun (acreate f)) = b.
Lemma bijective_target_aux: forall a b (f:a->b),
  target (inverse_fun (acreate f)) = a.
```

Let f be a function from A to B. We have shown before $x \subset f^{-1}\langle f\langle x\rangle\rangle$, if $x \subset$ A (this is true for any correspondence). Equality holds if f is injective. We have $f\langle f^{-1}\langle y\rangle\rangle \subset y$ if $y \subset$ B. Equality holds if f is surjective.

```
Lemma sub_inv_im_source:forall f y,
  is_function f -> sub y (target f) ->
  sub (inv_image_by_graph (graph f) y) (source f).

Lemma direct_inv_im: forall f y,
  is_function f ->  sub y (target f) ->
  sub (image_by_fun f (image_by_fun (inverse_fun f) y)) y.

Lemma direct_inv_im_surjective: forall f y,
  surjective  f -> sub y (target f) ->
  (image_by_fun f (image_by_fun (inverse_fun f) y)) = y.

Lemma inverse_direct_image:forall f x,
  is_function f -> sub x (source f) ->
  sub x (image_by_fun (inverse_fun f) (image_by_fun f x)).
Qed.

Lemma inverse_direct_image_inj:forall f x,
  injective f -> sub x (source f) ->
   x = (image_by_fun (inverse_fun f) (image_by_fun f x)).
```

4.8 Retractions and sections

$$\mathrm{A} \xrightarrow{f} \mathrm{B} \xrightarrow{r} \mathrm{A} \quad (\mathrm{I_A}) \qquad\qquad \mathrm{B} \xrightarrow{s} \mathrm{A} \xrightarrow{f} \mathrm{B} \quad (\mathrm{I_B}) \qquad\qquad \text{(retraction/section)}$$

A *retraction* r of f is a right inverse; a *section* s is a left inverse. This means that $r \circ f$ and $f \circ s$ are the identity functions. Assume f is a function from A to B. The definition of r implies the existence of $r \circ f$, i.e. the source of r is B. A consequence is that the target is A. In the same way, the definition of s implies the existence of $f \circ s$, i.e. the target of s is A. A consequence is that the source is B. In the case of Coq functions, if f has type $a \to b$, its inverse r or s has type $b \to a$ (there is a unique type for r compatible with the relation $r \circ f = \mathrm{I_A}$).

```
Definition is_left_inverse r f :=
  composable r f & compose r f = identity_fun (source f).

Definition is_right_inverse s f :=
  composable f s & compose f s = identity_fun (target f).

Definition is_left_inverseC (a b:Set) r (f:a->b) := composeC r f = identityC a.
Definition is_right_inverseC (a b:Set) s (f:a->b):= composeC f s = identityC b.

Lemma target_left_inverse: forall r f,
  is_left_inverse r f ->  target r = source f.
Lemma source_right_inverse: forall s f,
  is_right_inverse s f ->  source s = target f.
Lemma W_right_inverse: forall s f x,
  is_right_inverse s f -> inc x (target f) -> W (W x s) f = x.
Lemma W_left_inverse: forall r f x,
  is_left_inverse r f -> inc x (source f) -> W (W x f) r = x.
Lemma w_right_inverse: forall (a b:Set) s (f:a->b) (x:b),
  is_right_inverseC s f ->  f (s x) = x.
Lemma w_left_inverse: forall (a b:Set) r (f:a->b) (x:a),
  is_left_inverseC r f -> r (f x) = x.
```

Proposition 8 [2, p. 86] expresses the next four theorems. Assume that f is a function from A to B. If for some function s, $f \circ s = \mathrm{I_B}$ then f is surjective; if for some function r, $r \circ f = \mathrm{I_A}$ then f is injective. The converse holds; one has to take care that if $\mathrm{A} = \emptyset$, every function is injective, and there is in general no function from B to A (unless B is empty). Hence for the retraction r to exist, we assume $\mathrm{A} \neq \emptyset$. We start with the easy case.

```
Lemma inj_if_exists_left_invC: forall (a b:Set) (f:a-> b),
  (exists r, is_left_inverseC r f) -> injectiveC f.
Lemma surj_if_exists_right_invC: forall (a b:Set) (f:a->b),
  (exists s, is_right_inverseC s f) -> surjectiveC f.

Theorem inj_if_exists_left_inv: forall f,
  (exists r, is_left_inverse r f) -> injective f.
Theorem surj_if_exists_right_inv: forall f,
  (exists s, is_right_inverse s f) -> surjective f.
```

We show then existence of left and right inverses for Coq functions as follows. Assume that $f : a \to b$ is surjective. This means that for every $x : b$ there exists $y : a$ such that $f(y) = x$. We use *chooseT*, the basic axiom of choice to select such an y. For this mechanism to work, we need a proof that a is not the empty type; this is only required when we try to assign a value to x, and the existence of y is then sufficient.

Assume f injective and $w : a$. Given $x : b$, we consider $\mathrm{P}(y)$ the property that "$f(y) = x$ or x is not in the image of f", and $y = w$. Such an y exists, and we can apply the axiom of choice. If $x = f(z)$, then we are in the first case, hence $f(y) = f(z)$. Denote the inverse mapping by g. We have $f(g(f(z))) = f(z)$ for every z. The assumption that f is injective says $g(f(z)) = z$, so that g is a left inverse.

```
Lemma left_inverseC_aux: forall (a b:Set) (f: a->b)(w:a) (v:b),
  exists u:a, (~ (exists x:a, f x = v) & u  = w)  \/ (f u = v).
Definition chooseT_any a (H:nonemptyT a):= (chooseT (fun x:a => x=x) H).

Definition left_inverseC (a b:Set) (f: a->b)(H:nonemptyT a)
```

```
  (v:b) := (chooseT (fun u:a => (~ (exists x:a, f x = v) & u  = chooseT_any H)
    \/ (f u = v)) H).

Lemma left_inverseC_pr:forall (a b:Set) (f: a->b) (H:nonemptyT a) (u:a),
  f(left_inverseC f H (f u)) = f u.
Lemma left_inverse_comp_id: forall (a b:Set) (f:a->b)  (H:nonemptyT a),
  injectiveC f ->  composeC (left_inverseC f H) f = identityC a.
Lemma exists_left_inv_from_injC: forall (a b:Set) (f:a->b),nonemptyT a ->
  injectiveC f -> exists r, is_left_inverseC r f.

Lemma inverse_value_ex: forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  nonemptyT a.
Definition right_inverseC (a b:Set) (f: a->b)  (H:surjectiveC f) (x:b) :=
  (chooseT (fun k:a => f k = x) (inverse_value_ex H x)).
Lemma right_inverse_pr:  forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  f(right_inverseC H x) = x.
Lemma right_inverse_pr:  forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  f(right_inverseC H x) = x.
Lemma right_inverse_comp_id: forall (a b:Set) (f:a-> b) (H:surjectiveC f),
  composeC f (right_inverseC H)  = identityC b.
Lemma exists_right_inv_from_surjC: forall (a b:Set) (f:a-> b)(H:surjectiveC f),
  exists s, is_right_inverseC s f.
```

Bourbaki shows existence of a left inverse of the function $f : A \to B$ by considering the subset of $B \times A$ formed of all pairs (x, y) such that $y \in A$ and $y = f(x)$ or $y = e$ and $x \in B \setminus f\langle A \rangle$, where $e \in A$ (such an element exists when A is nonempty). This set is a functional graph, and the function with this graph is an answer to the question.

```
Theorem exists_left_inv_from_inj: forall f,
  injective f ->nonempty (source f) -> exists r:E, is_left_inverse r f.
Theorem exists_right_inv_from_surj: forall f,
  surjective  f -> exists s:E, is_right_inverse s f.
Theorem exists_left_inv_from_inj_alt: forall f,
  injective f -> nonempty (source f) -> exists r, is_left_inverse r f.  (* 41 *)
Theorem exists_right_inv_from_surj_alt: forall f,
  surjective  f -> exists s, is_right_inverse s f.   (* 17 *)
```

¶ Some consequences. If r is a left inverse of f, then f is a right inverse of r, and vice versa. A left inverse is surjective, a right inverse is injective. If g is both a left inverse and a right inverse of f, then g is bijective as well as f.

```
Lemma bijective_from_compose: forall g f,
  composable g f -> composable f g -> compose g f = identity_fun (source f)
  -> compose f g = identity_fun (source g)
  ->(bijective f & bijective g & g = inverse_fun f).   (* 16 *)

Lemma right_inverse_from_leftC: forall (a b:Set) (r:b->a)(f:a->b),
  is_left_inverseC r f -> is_right_inverseC f r.
Lemma left_inverse_from_rightC: forall (a b:Set) (s:b->a)(f:a->b),
  is_right_inverseC s f -> is_left_inverseC f s.
Lemma left_inverse_surjectiveC: forall (a b:Set) (r:b->a)(f:a->b),
  is_left_inverseC r f -> surjectiveC r.
Lemma right_inverse_injectiveC: forall (a b:Set) (s:b->a)(f:a->b),
  is_right_inverseC s f -> injectiveC s.
Lemma section_uniqueC: forall (a b:Set) (f:a->b)(s:b->a)(s':b->a),
```

```
  is_right_inverseC s f -> is_right_inverseC s' f ->
  (forall x:a, (exists u:b, x = s u) = (exists u':b, x = s' u')) ->
  s = s'.

Lemma right_inverse_from_left: forall r f,
  is_left_inverse r f -> is_right_inverse f r.
Lemma left_inverse_from_right: forall s f,
  is_right_inverse s f -> is_left_inverse f s.
Lemma left_inverse_surjective: forall f r,
  is_left_inverse r f -> surjective r.
Lemma right_inverse_injective: forall f s,
  is_right_inverse s f -> injective s.
Lemma section_unique: forall f s s',
  is_right_inverse s f -> is_right_inverse s' f ->
  range (graph s) = range (graph s') ->s = s'.
```

Theorem 1 in Bourbaki [2, p. 87] comes next. We assume that f and f' are two composable functions and $f'' = f' \circ f$. If f and f' are injective so is f'', if f and f' are surjective, so is f''. Hence, if f and f' are bijections, so is f''. If f and f' have a left inverse, so has f'' (it is the composition of the inverses in reverse order). The same holds for right inverses.

If f'' has a left inverse r'' then $r'' \circ f'$ is a right inverse of f, and $f \circ r''$ is a left inverse of f' provided that f is surjective (in which case f is invertible). If f'' has a right inverse s'' then $f \circ s''$ is a right inverse of f' and $s'' \circ f'$ is a left inverse of f, provided that f' is injective, in which case f' is a bijection.

If f'' is injective then f is injective, and for f' to be injective it suffices that f is surjective; if f'' is surjective then f' is surjective; and for f to be surjective it suffices that f' is injective.

```
Lemma inj_composeC: forall (a b c:Set) (f:a->b)(f':b->c),
  injectiveC f-> injectiveC f' -> injectiveC (composeC f' f).
Lemma surj_composeC: forall (a b c:Set) (f:a->b)(f':b->c),
  surjectiveC f-> surjectiveC f' -> surjectiveC (composeC f' f).
Lemma left_inverse_composeC: forall (a b c:Set)
  (f:a->b) (f':b->c)(r:b->a)(r':c->b),
  is_left_inverseC r' f' -> is_left_inverseC r f ->
  is_left_inverseC (composeC r r') (composeC f' f).
Lemma right_inverse_composeC: forall (a b c:Set)
  (f:a->b) (f':b->c)(s:b->a)(s':c->b),
  is_right_inverseC s' f' -> is_right_inverseC s f ->
  is_right_inverseC (composeC s s') (composeC f' f).
Lemma inj_right_composeC: forall (a b c:Set) (f:a->b) (f':b->c),
  injectiveC (composeC f' f) -> injectiveC f.
Lemma surj_left_composeC: forall (a b c:Set) (f:a->b) (f':b->c),
  surjectiveC (composeC f' f) -> surjectiveC f'.
Lemma surj_left_compose2C: forall (a b c:Set) (f:a->b) (f':b->c),
  surjectiveC (composeC f' f) -> injectiveC f' -> surjectiveC f.
Lemma inj_left_compose2C: forall (a b c :Set)(f:a->b) (f':b->c),
  injectiveC (composeC f' f) -> surjectiveC f -> injectiveC f'.
Lemma left_inv_compose_rfC: forall (a b c:Set) (f:a->b) (f':b->c)(r'': c->a),
  is_left_inverseC r'' (composeC f' f) ->
  is_left_inverseC (composeC r'' f') f.
Lemma right_inv_compose_rfC :  forall (a b c:Set) (f:a->b) (f':b->c)(s'': c->a),
  is_right_inverseC s'' (composeC f' f) ->
  is_right_inverseC (composeC f s'') f'.
Lemma left_inv_compose_rf2C: forall (a b c:Set) (f:a->b) (f':b->c)(r'': c->a),
```

```
  is_left_inverseC r'' (composeC f' f) -> surjectiveC f ->
  is_left_inverseC (composeC f r'') f'.
Lemma right_inv_compose_rf2C : forall (a b c:Set) (f:a->b) (f':b->c)(s'': c->a),
  is_right_inverseC s'' (composeC f' f) -> injectiveC f'->
  is_right_inverseC (composeC s'' f') f.
```

Now the same results, in Bourbaki notations.

```
Theorem inj_compose: forall f f',
  injective f-> injective f' -> composable f' f ->
  injective (compose f' f).
Theorem surj_compose: forall f f',
  surjective f-> surjective f' -> composable f' f ->
  surjective (compose f' f).
Lemma bij_compose: forall f f',
  bijective f-> bijective f' -> composable f' f ->
  bijective (compose f' f).
Lemma left_inverse_composable: forall f f' r r',  composable f' f ->
  is_left_inverse r' f' -> is_left_inverse r f -> composable r r'.
Lemma right_inverse_composable: forall f f' s s',  composable f' f ->
  is_right_inverse s' f' -> is_right_inverse s f -> composable s s'.
Theorem left_inverse_compose: forall f f' r r', composable f' f ->
  is_left_inverse r' f' -> is_left_inverse r f ->
  is_left_inverse (compose r r') (compose f' f).
Theorem right_inverse_compose: forall f f' s s', composable f' f ->
  is_right_inverse s' f' -> is_right_inverse s f ->
  is_right_inverse (compose s s') (compose f' f).
Theorem inj_right_compose: forall f f',
  composable f' f-> injective (compose f' f) -> injective f.
Theorem surj_left_compose: forall f f',
  composable f' f-> surjective (compose f' f) -> surjective f'.
Theorem left_inv_compose_rf: forall f f' r'',
  composable f' f-> is_left_inverse r'' (compose f' f) ->
  is_left_inverse (compose r'' f') f.
Theorem right_inv_compose_rf : forall f f' s'',
  composable f' f-> is_right_inverse s'' (compose f' f) ->
  is_right_inverse (compose f s'') f'.
Theorem surj_left_compose2: forall f f',
  composable f' f-> surjective (compose f' f) -> injective f' -> surjective f.
Theorem inj_left_compose2: forall f f',
  composable f' f-> injective (compose f' f) -> surjective f -> injective f'.
Theorem left_inv_compose_rf2: forall f f' r'',
  composable f' f-> is_left_inverse r'' (compose f' f) -> surjective f ->
  is_left_inverse (compose f r'') f'.    (* 25 *)
Theorem right_inv_compose_rf2 : forall f f' s'',
  composable f' f-> is_right_inverse s'' (compose f' f) -> injective f'->
  is_right_inverse (compose s'' f') f.    (* 24 *)
```

If $f \circ g$ is a bijection, one function is a bijection, so is the other.

```
Lemma bij_right_compose: forall f f',
  composable f' f-> bijective (compose f' f) -> bijective f' ->bijective f.
Lemma bij_left_compose: forall f f',
  composable f' f-> bijective (compose f' f) -> bijective f ->bijective f'.
```

Next three lemmas show that equipotency is an equivalence relation.

```
Lemma equipotent_reflexive: forall x, equipotent x x.
Lemma equipotent_symmetric:  forall a b,
  equipotent a b -> equipotent b a.
Lemma equipotent_transitive: forall a b c,
  equipotent a b -> equipotent b c -> equipotent a c.
```

(decomposition, Prop 9)

Proposition 9 [2, p. 88] is implemented in the next lemmas. If f and g have the same source and if g is surjective, then the condition $g(x) = g(y) \implies f(x) = f(y)$ is a necessary and sufficient condition for the existence of h with $f = h \circ g$. Such a mapping is then unique and is $f \circ s$, for any right inverse of g.

```
Lemma exists_left_composableC: forall (a b c:Set) (f:a->b)(g:a->c),
  surjectiveC g ->
  (exists h, composeC h g = f) =
  (forall (x y:a), g x = g y -> f x = f y).
Theorem exists_left_composable: forall f g,
  is_function f -> surjective g -> source f = source g ->
  (exists h:E, composable  h g & compose h g = f) =
  (forall (x y:E), inc x (source g) -> inc y (source g) ->
    W x g = W y g -> W x f = W y f).  (* 13 *)

Lemma exists_left_composable_auxC: forall (a b c:Set) (f:a->b) (g:a-> c) s h,
  surjectiveC g -> is_right_inverseC s g ->
  composeC h g = f -> h = composeC f s.
Theorem exists_left_composable_aux: forall f g s h ,
  is_function f -> surjective g -> source f = source g ->
  is_right_inverse s g ->
  composable  h g -> compose h g = f -> h = compose f s.

Lemma exists_unique_left_composableC: forall (a b c:Set) (f:a->b)(g:a->c) h h',
  surjectiveC g -> composeC h g = f -> composeC h' g = f ->
  h = h'.
Theorem exists_unique_left_composable: forall f g h h',
  is_function f -> surjective g -> source f = source g ->
  composable  h g -> compose h g = f ->
  composable  h' g -> compose h' g = f -> h = h'.

Lemma left_composable_valueC: forall (a b c:Set) (f:a->b)(g:a->c) s h,
  surjectiveC g ->  (forall (x y:a), g x = g y -> f x = f y) ->
  is_right_inverseC s g  ->  h = composeC f s ->
  composeC h g = f.
Theorem left_composable_value: forall f g s h,
  is_function f -> surjective g -> source f = source g ->
  (forall (x y:E), inc x (source g) -> inc y (source g) ->
    W x g = W y g -> W x f = W y f) ->
  is_right_inverse s g  ->   h = compose f s-> compose h g = f.
```

Second part of Proposition 9. We assume that f and g have the same target, g is injective; the condition range$(f) \subset$ range(g) is a necessary and sufficient condition for the existence of h with $f = g \circ h$, such a mapping is then unique and is $r \circ f$, for any left inverse of g.

```
Lemma exists_right_composable_auxC: forall (a b c:Set) (f:a->b) (g:c->b) h r,
  injectiveC g ->  is_left_inverseC r g  -> composeC g h = f
  -> h = composeC r f.
Theorem exists_right_composable_aux: forall f g h r,
  is_function f -> injective g -> target f = target g ->
  is_left_inverse r g  -> composable g h ->  compose g h = f
  -> h = compose r f.

Lemma exists_right_composable_uniqueC: forall (a b c:Set) (f:a->b)(g:c->b) h h',
  injectiveC g -> composeC g h = f -> composeC g h' = f -> h = h'.
Theorem exists_right_composable_unique: forall f g h h',
  is_function f -> injective g -> target f = target g ->
  composable g h ->  compose g h = f ->
  composable g h' ->  compose g h' = f -> h = h'.

Lemma exists_right_composableC: forall (a b c:Set) (f:a->b) (g:c->b),
  injectiveC g ->
  (exists h, composeC g h = f) =  (forall u, exists v, g v = f u).
Theorem exists_right_composable: forall f g,
  is_function f -> injective g -> target f = target g ->
  (exists h, composable  g h & compose g h = f) =
  (sub (range (graph f)) (range (graph g))).   (* 41 *)

Lemma right_composable_valueC: forall (a b c:Set) (f:a->b) (g:c->b) r h,
  injectiveC g -> is_left_inverseC r g -> (forall u, exists v, g v = f u) ->
  h = composeC r f ->  composeC g h = f.    (* 17 *)
Theorem right_composable_value: forall f g r h,
  is_function f -> injective g -> target g = target f ->
  is_left_inverse r g ->
  (sub (range (graph f)) (range (graph g))) ->
  h = compose r f ->,
  compose g h = f.
```

4.9 Functions of two arguments

By definition, all functions take one argument. Consider for example addition of integers. Its type is $nat \to nat \to nat$, so that addition is a function that takes an argument a, and returns a function that takes an argument b and returns the sum $a+b$. An other example is the pair constructor of type $A \to B \to A * B$. We can consider the function of type $nat * nat \to nat$, that associates to a pair (a,b) the sum $a+b$. This may also be called a function of two arguments. Switching betweeen these two points of view is called currying or uncurrying.

An operator of type $A \to B \to C$ will be called a function of two arguments; for instance the union of two sets (where all three types are Set), or composition of function (where the types are *correspondenceC*). When we consider functions in the Bourbaki sense (with a source, a target and a graph), the second point of view will be used. More precisely, we interpret $A \to B$ to mean that the source is A and the target is B. A function of two arguments of type $A * B \to C$, is thus a function whose source is $A \times B$ and its target is C. For each element of A we get a function $B \to C$. Similarly, for each element of B we get a function $A \to C$. These two definitions of a function of two arguments are equivalent, it will be formalized in Chapter 6.

```
Definition partial_fun2 f y :=
  BL(fun x=> W (J x y) f) (im_singleton(inverse_graph (source f)) y) (target f).
```

```
Definition partial_fun1 f x :=
  BL(fun y=> W (J x y) f)(im_singleton (source f) x) (target f).

Lemma axioms_partial_fun1: forall f x, is_function f -> is_graph(source f) ->
  transf_axioms (fun y=> W (J x y) f)(im_singleton (source f) x) (target f).
Lemma function_partial_fun1: forall f x, is_function f -> is_graph(source f) ->
  is_function (partial_fun1 f x).
Lemma W_partial_fun1: forall f x y, is_function f -> is_graph(source f) ->
  inc (J x y) (source f) -> W y (partial_fun1 f x) = W (J x y) f.
Lemma axioms_partial_fun2: forall f y, is_function f -> is_graph(source f) ->
  transf_axioms (fun x=> W (J x y) f)(im_singleton(inverse_graph (source f)) y)
  (target f).
Lemma function_partial_fun2: forall f y, is_function f -> is_graph(source f) ->
  is_function (partial_fun2 f y).
Lemma W_partial_fun2: forall f x y, is_function f -> is_graph(source f) ->
  inc (J x y) (source f) -> W x (partial_fun2 f y) = W (J x y) f.
```

An example of function of two arguments is the function obtained from two functions u and v by associating to (x, y) the pair $(u(x), v(y))$.

```
Definition ext_to_prod u v :=
  BL(fun z=> J (W (P z) u)(W (Q z) v))
  (product (source u)(source v))
  (product (target u)(target v)).
Lemma function_ext_to_prod: forall u v,
  is_function  u -> is_function v ->
  is_function (ext_to_prod u v).
Lemma W_ext_to_prod: forall u v a b,
  is_function u -> is_function v-> inc a (source u) -> inc b (source v)->
  W (J a b) (ext_to_prod u v)  =  J (W a u) (W b v).
Lemma W_ext_to_prod2: forall u v c,
  is_function u -> is_function v->
  inc c (product (source u)(source v)) ->
Lemma range_ext_to_prod2: forall u v,
  is_function u -> is_function v->
  range (graph (ext_to_prod u v)) =
    product (range (graph u))(range (graph v)).  (* 14 *)
```

$$\begin{array}{ccccc} a & \xleftarrow{\text{pr1C}} & a\times b & \xrightarrow{\text{pr2C}} & b \\ \downarrow{\scriptstyle u} & & \downarrow{\scriptstyle u\times v} & & \downarrow{\scriptstyle v} \\ a' & \xrightarrow[\text{J}]{} & a'\times b' & \xleftarrow[\text{J}]{} & b' \end{array} \qquad \text{(prod extension)}$$

We can consider the product of two Coq functions. We first define the projections from $a \times b$ to a and b and the inverse function. In the diagram above, this inverse function corresponds to the two arrows named J. In other words, if z is the pair (x, y), we have $\mathrm{P}(z) = x$ and $\mathrm{Q}(z) = y$. To say that J is the inverse means that J applied to x and y gives z. This function takes two arguments (its type is $\mathscr{E} \to \mathscr{E} \to \mathscr{E}$) but is not a function of two arguments (its type is not $\mathscr{E} \times \mathscr{E} \to \mathscr{E}$).

```
Lemma ext_to_prod_propP: forall a a' (x: product a a'),  inc (P (Ro x)) a.
Lemma ext_to_prod_propQ: forall a a' (x: product a a'),  inc (Q (Ro x)) a'.
```

```
Lemma ext_to_prod_propJ: forall (b b':Set) (x:b)(x':b'),
  inc (J (Ro x)(Ro x'))  (product b b').


Definition pr1C a b:= fun x:product a b => Bo(ext_to_prod_propP x).
Definition pr2C a b:= fun x:product a b => Bo(ext_to_prod_propQ x).
Definition pairC a b:= fun (x:a)(y:b) => Bo(ext_to_prod_propJ x y).

Definition ext_to_prodC (a b a' b':Set) (u:a->a')(v:b->b') :=
  fun x => pairC (u (pr1C x)) (v (pr2C x)).

Lemma prC_prop: forall (a b:Set) (x:product a b),
  Ro x = J (Ro (pr1C x)) (Ro (pr2C x)).
Lemma pr1C_prop: forall (a b:Set) (x:product a b),  Ro (pr1C x) =  P (Ro x).
Lemma pr2C_prop: forall (a b:Set) (x:product a b),  Ro (pr2C x) =  Q (Ro x).
Lemma prJ_prop: forall (a b:Set) (x:a)(y:b),   Ro(pairC x y) = J (Ro x) (Ro y).
Lemma prJ_recov: forall (a b:Set) (x:product a b), pairC (pr1C x) (pr2C x) = x.
Lemma ext_to_prod_prop:
  forall (a b a' b':Set) (u:a->a')(v:b->b') (x:a)(x':b),
  J(Ro (u x)) (Ro (v x')) = Ro(ext_to_prodC u v (pairC x x')).
```

If both functions are injective, surjective or bijective, so is the product. The inverse is the product of the inverses. It is compatible with composition.

```
Lemma injective_ext_to_prod2: forall u v,
  injective u -> injective  v-> injective (ext_to_prod u v).
Lemma surjective_ext_to_prod2: forall u v,
  surjective u -> surjective  v-> surjective (ext_to_prod u v).
Lemma bijective_ext_to_prod2: forall u v,
  bijective u -> bijective  v-> bijective (ext_to_prod u v).
Lemma inverse_ext_to_prod2: forall u v,
  bijective u -> bijective  v->
  inverse_fun (ext_to_prod u v) =
  ext_to_prod (inverse_fun u)(inverse_fun v). (* 17 *)
Lemma composable_ext_to_prod2: forall u v u' v',
  composable u' u -> composable v' v ->
  composable (ext_to_prod u' v') (ext_to_prod u v).
Lemma compose_ext_to_prod2: forall u v u' v',
  composable u' u ->   composable v' v ->
  compose (ext_to_prod u' v') (ext_to_prod u v) =
  ext_to_prod (compose u' u)(compose v' v).  (* 19 *)
```

Same lemmas for Coq functions.

```
Lemma injective_ext_to_prod2C: forall (a b a' b':Set) (u:a->a')(v:b->b'),
  injectiveC u -> injectiveC  v-> injectiveC (ext_to_prodC u v).
Lemma surjective_ext_to_prod2C: forall (a b a' b':Set) (u:a->a')(v:b->b'),
  surjectiveC u -> surjectiveC  v-> surjectiveC (ext_to_prodC u v).
Lemma bijective_ext_to_prod2C:  forall (a b a' b':Set) (u:a->a')(v:b->b'),
  bijectiveC u -> bijectiveC  v-> bijectiveC (ext_to_prodC u v).
Lemma compose_ext_to_prod2C: forall (a b c a' b' c':Set) (u:b-> c)(v:a->b)
  (u':b'-> c')(v':a'->b'),
  composeC (ext_to_prodC u u') (ext_to_prodC v v') =
  ext_to_prodC (composeC u v)(composeC u' v').
Lemma inverse_ext_to_prod2C: forall (a b a' b':Set) (u:a->a')(v:b->b')
  (Hu: bijectiveC u)(Hv:bijectiveC  v),
```

```
  inverseC (bijective_ext_to_prod2C Hu Hv)=
  ext_to_prodC (inverseC Hu)(inverseC Hv).
```

¶ Canonical decomposition of a function, version one. Let f be a function from A to B, and C its range. Then f is the composition of the restriction of f to its range, and the canonical injection from the range to the target. The first function satisfies $g(x) = f(x)$; the second satisfies $i(x) = x$.

```
Lemma canonical_decomposition1: forall f g i,
  is_function f ->
  g = restriction2 f (source f) (range (graph f)) ->
  i = canonical_injection (range (graph f)) (target f) ->
  (composable i g & f = compose i g & injective i & surjective g &
  (injective f -> bijective g )).  (* 21 *)
```

In the case of Coq functions, we replace the range of the graph by the image.

```
Definition imageC (a b:Set) (f:a->b) := IM (fun u:a => Ro (f u)).
Lemma imageC_inc: forall (a b:Set) (f:a->b) (x:a), inc (Ro (f x)) (imageC f).
Lemma imageC_exists: forall (a b:Set) (f:a->b) x,
  inc x (imageC f) -> exists y:a,  x = Ro (f y).
Lemma sub_image_targetC: forall (a b:Set) (f:a->b), sub (imageC f) b.

Definition restriction_to_image a b (f:a->b) :=
  fun x:a => Bo (imageC_inc f x).

Lemma restriction_to_image_pr: forall (a b:Set) (f:a->b) (x:a),
  Ro(restriction_to_image f x) = Ro (f x).
Lemma canonical_decomposition1C: forall (a b:Set) (f:a->b)
  (g:a-> imageC f)(i:imageC f ->b),
  g = restriction_to_image f ->
  i = inclusionC (sub_image_target (f:=f))  ->
  (injectiveC i & surjectiveC g &
  (injectiveC f -> bijectiveC g )).
```

Chapter 5

Union and intersection of a family of sets

Bourbaki defines union, intersection and products of a family of sets. A family is just a function, that is, a source, a target, and a functional graph. The definition of the union [2, p. 90] is: "let $(X_\iota)_{\iota\in I}$ be a family of sets. The set [...] is called the union of the family and denoted by $\bigcup_{\iota\in I} X_\iota$." Intersection is similarly denoted by $\bigcap_{\iota\in I} X_\iota$. In this notation ι is a dummy variable, X is the graph, I the source (called "the index set"), and the target is never mentioned. In fact, the definitions are independent of the target. Given a graph G, it is possible to construct a function with graph G (just use domain and range as source and target). Note that we do not have any choice for the source. The associated union will be independent of these choices.

For this reason, in what follows, *unionb X* is the union associated to the graph X. Using *unionb* requires that X be a functional graph. Assume now that I is a set, and X a mapping. Then $\mathscr{L}_I X$ is a functional graph, whose index set is I. For this reason, we introduce *unionf I X*. This better matches the definition of Bourbaki. In some cases, Bourbaki considers the family $(X_{f(\kappa)})_{\kappa\in K}$. This has to be understood as the family $X \circ f$.

We consider another variant, *uniont* where X is a function on the type I with values into a set. This will be our primary definition. We must then show that all these definitions are equivalent, and the same as the original *union* (as defined by C. Simpson in Sections 2.9 and 2.10).

Intersection is only defined over a nonempty index set. We have tried to impose this restriction in the definition, but this gives theorems that are two complicated. As a result, we define intersection even over the empty set; the situation is a a bit complicated, since we define intersection over any type, and we have a representative only if the type is non-empty.

5.1 Definition of the union and intersection of a family of sets

We define here *uniont f*, *unionf I X* and *unionb G* as follows. The first definition is a variant of the union, as defined in section 2.9. A *Uintegral* record contains z and e of type $f(z)$, so that $\mathscr{R}e \in f(z)$. Hence the union is just the image of the function that associates to each record the quantity $\mathscr{R}e$.

The intersection of a function f defined on a type I, denoted by *intersectiont f*, is the set of all $y \in f(a)$ such that $y \in f(z)$ for all z; it is independent of a. We choose for a a representative

of I; this can be done if I is not empty, otherwise, the intersection is defined to be the empty set.

Given a set I and a mapping X defined on sets, we define *unionf I X* and *intersectionf I X* as the union and intersection of the functions defined on the type I by composing X with $\mathscr{R}$. If G is a graph, we define *unionb G* and *intersection G* as *unionf I X* and *intersectionf I X*, where I is the domain and X the evaluation function. It will be shown that *intersection X*, where X is a set of sets, is the intersection of the identity function on X. Similarly, *union X* is the union of the identity function.

```
Record Uintegral (In :Set)(f :In->Set) : Set := {UI_z : In; UI_elt : f UI_z}.

Definition uniont (In:Set)(f : In->Set) :=
   IM (fun i : Uintegral f => Ro (UI_elt i)).

Definition intersectiont (In:Set)(f : In->Set):=
  by_cases(fun H:nonemptyT In =>
      Zo (f (chooseT_any H)) (fun y => forall z : In, inc y (f z)))
  (fun _:~ nonemptyT In => emptyset).

Definition unionf (x:Set)(f: Set->Set) := uniont (fun a:x => f (Ro a)).
Definition unionb (g:Set) := unionf (domain g)(fun a=> V a g).
Definition intersectionf (x:Set)(f: Set->Set):=
  intersectiont(fun a:x => f (Ro a)).
Definition intersectionb (g:Set) :=
  intersectionf (domain g) (fun a=> V a g).
```

We have now a bunch of lemmas that show how to use these definitions.

```
Lemma uniont_rw: forall (In:Set) (f:In->Set) x,
  inc x (uniont f) = exists z,  inc x (f z).
Lemma unionf_rw: forall x i f,
  inc x (unionf i f) =  exists y, inc y i & inc x (f y).
Lemma unionb_rw: forall x f,
  inc x (unionb f) =  exists y, inc y (domain f) & inc x (V y f).
Lemma uniont_inc : forall (In:Set) (f : In->Set) y x,
  inc x (f y) -> inc x (uniont f).
Lemma uniont_exists : forall (In:Set) (f : In->Set) x,
  inc x (uniont f) -> exists y:In, inc x (f y).
Lemma unionf_inc: forall x y i f,
  inc y i -> inc x (f y) -> inc x (unionf i f).
Lemma unionf_exists:forall x i f,
  inc x (unionf i f) ->  exists y, inc y i & inc x (f y).
Lemma unionb_inc: forall x y f,
  inc y (domain f) -> inc x (V y f) ->  inc x (unionb f).
Lemma unionb_exists: forall x f,
  inc x (unionb f) -> exists y, inc y (domain f) & inc x (V y f).
```

Same lemmas for the intersection. All these lemmas are obvious from the definitions and the link between $\mathscr{R}$ and $\mathscr{B}$.

```
Lemma intersectiont_rw: forall (In:Set) (f:In-> Set) x,
  nonemptyT In ->
  inc x (intersectiont f) = (forall j, inc x (f j)).
Lemma intersectionf_rw : forall In f x,
```

```
  nonempty In ->
  inc x (intersectionf In f) = (forall j, inc j In -> inc x (f j)).
Lemma intersectionb_rw : forall g x,
  nonempty g ->
  inc x (intersectionb g) = (forall i inc i (domain g) -> inc x (V i g)).
Lemma intersectiont_inc : forall (In:Set) (f:In-> Set) x,
  nonemptyT In ->
  (forall j, inc x (f j)) -> inc x (intersectiont f).
Lemma intersectiont_forall : forall (In:Set) (f:In-> Set) x j,
  inc x (intersectiont f) -> inc x (f j).
Lemma intersectionf_inc :forall In f x,
  nonempty In ->
  (forall j, inc j In -> inc x (f j)) -> inc x (intersectionf In f).
Lemma intersectionf_forall :forall In f x j,
  inc x (intersectionf In f) -> inc j In -> inc x (f j).
Lemma intersectionb_inc : forall g x,
  nonempty g ->
  (forall i, inc i (domain g) -> inc x (V i g)) -> inc x (intersectionb g).
Lemma intersectionb_forall : forall g x i,
  inc x (intersectionb g) -> inc i (domain g) -> inc x (V i g).
```

These lemmas are trivial consequences of the previous ones. They explain when two unions or intersections are equal.

```
Lemma uniont_extensionality: forall (In:Set) (f: In-> Set) (f':In->Set),
  (forall i, f i = f' i) -> uniont f = uniont f'.
Lemma unionf_extensionality: forall sf f f',
  (forall i, inc i sf -> f i = f' i) -> unionf sf f = unionf sf f'.
Lemma unionb_extensionality: forall f f',
  f = f' -> unionb f = unionb f'.
Lemma intersectiont_extensionality:
  forall (In:Set) (f:In-> Set) (f':In-> Set), nonemptyT In ->
    (forall i, f i = f' i) -> (intersectiont f) = (intersectiont f').
Lemma intersectionf_extensionality:  forall In f f, nonempty In ->
  (forall i, inc i In -> f i = f' i) ->
  intersectionf In f  = intersectionf In f'.
Lemma intersectionb_extensionality: forall g g',
  g = g' -> intersectionb g = intersectionb g'.
```

These trivial lemmas say that for all j, $X_j \subset \bigcup X_i$ and $\bigcap X_i \subset X_j$. On the other hand, if for all i, we have $A \subset X_i \subset B$, then $A \subset \bigcup X_i \subset B$ and $A \subset \bigcap X_i \subset B$. Note that for two of these inclusions, the index set must be nonempty.

```
Lemma uniont_sub:  forall (In:Set) (f: In-> Set) i,
  sub (f i) (uniont f).
Lemma intersectiont_sub:  forall (In:Set) (f: In-> Set) i,
  nonemptyT In -> sub (intersectiont f) (f i).
Lemma sub_uniont:  forall (In:set) (f: In-> Set) x,
  (forall i, sub (f i) x) ->  sub (uniont f) x.
Lemma sub_intersectiont:  forall (In:Set) (f: In-> Set) x,
  nonemptyT In -> (forall i, sub x (f i)) ->  sub x (intersectiont f).
Lemma intersectiont_sub2:  forall (In:Set) (f: In-> Set) x,
   nonemptyT In -> (forall i, sub (f i) x) ->  sub (intersectiont f) x.
Lemma sub_uniont2:  forall (In:Set) (f: In-> Set) x,
  nonemptyT In -> (forall i, sub x (f i)) -> sub x (uniont f).
```

If the index set is empty, so is the union.

```
Lemma empty_uniont1: forall (In:Set) (f: In-> Set),
  nonempty (uniont f) -> nonemptyT In.
Lemma empty_unionf1: forall sf f,
  nonempty (unionf sf f) -> nonempty sf.
Lemma empty_unionf: forall sf f,
  sf = emptyset -> unionf sf f = emptyset.
```

Bourbaki says in Proposition 1 [2, p. 92]: Let f be a function from K onto I, X_ι a family of sets indexed by I. Then the union and the intersection of the family is the union and the intersection of $X_{f(\kappa)}$ over K.

```
Theorem uniont_rewrite: forall (In K:Set) (f: K->In) (g:In ->Set),
  surjectiveC f ->
  uniont g = uniont (fun k:K => g(f k)).
Theorem intersectiont_rewrite: forall (In K:Set) (f: K->In) (g:In ->Set)
  (Ha:nonemptyT In)  (Hb:surjectiveC f),
  intersectiont g = intersectiont (fun k:K => g(f k)).
```

The Bourbaki statement about union is *unionb_rewrite1*. In the second lemma we just assume that f is a functional graph. Note that we dropped the requirement that g must be a functional graph (we use *gcompose* instead).

```
Lemma unionb_rewrite1: forall f g,
  is_function f -> fgraph g -> range (graph f) = domain g ->
  unionb g = unionb (fcompose g (graph f)).
Lemma unionb_rewrite: forall f g,
  fgraph f -> range f = domain g ->
  unionb g = unionb (gcompose g f).
Lemma intersectionb_rewrite: forall f g,
  fgraph f -> range f = domain g -> nonempty g ->
  intersectionb g = intersectionb (gcompose g f).
```

Let f be a constant function and $x \in$ I. Then the intersection and union of f on I is $f(x)$. This is trivial to show. But we shall follow Bourbaki: we first consider the case where I is a singleton. Then we shall prove that a constant function h can be written as $h = g \circ f$ where the image of f is a singleton, hence f is surjective, and the union (or intersection) of h is that of g. The conclusion is then obvious, since the source of g is a singleton.

```
Lemma uniont_constant_alt: forall (In:Set) (f:In ->Set) (x:In),
  is_constant_functionC f -> uniont f = f x.
Lemma intersectiont_constant_alt: forall (In:Set) (f:In ->Set) (x:In),
  is_constant_functionC f -> intersectiont f = f x.


Lemma singleton_type_inj: forall x (y:singleton x)(z:singleton x),   y = z.
Lemma uniont_singleton:forall (a;Set) (x:a) (f: singleton (Ro x) -> Set),
  uniont f = f (Bo (singleton_inc (Ro x))).
Lemma intersectiont_singleton:forall (a:Set) (x:a) (f: singleton (Ro x) -> Set),
  intersectiont f = f (Bo (singleton_inc (Ro x))).
Lemma unionf_singleton:forall f x,
  unionf (singleton x) f = f x.
Lemma intersectionf_singleton:forall f x,
  intersectionf (singleton x) f = f x.
```

```
Lemma constant_function_pr: forall (a:Set) (h:a->Set) (x:a),
  is_constant_functionC h ->
  exists f: a->singleton (Ro x), exists g:singleton (Ro x)->Set,
    (forall u:a, h u = g (f u)) & (g (Bo (singleton_inc (Ro x))) = h x).
Lemma uniont_constant: forall (In:Set) (f:In ->Set) (x:In),
  is_constant_functionC f -> uniont f = f x.
Lemma intersectiont_constant: forall (In:Set) (f:In ->Set) (x:In),
  is_constant_functionC f -> intersectiont f = f x.
```

¶ Link between these unions and intersections and the old ones: the union of a set of sets X is the union of the identity function on X. If f is a functional graph, its union is also the union of the range.

A trivial consequence concerns union and intersection of a singleton.

```
Lemma union_prop: forall x,  union x = unionf x (fun u => u).
Lemma intersection_prop: forall x, nonempty x ->
  intersection x = intersectionf x (fun u => u).
Lemma unionb_alt: forall f, fgraph f -> unionb f = union (range f).
Lemma unionb_identity: forall x, unionb (identity x) = union x.
Lemma union_singleton: forall x,  union (singleton x) = x.
Lemma intersection_singleton: forall x,  intersection (singleton x) = x.
```

5.2 Properties of union and intersection

We first show that the union and intersection of F over I are monotone with respect to the function and index.

```
Lemma union_monotone: forall (In:Set) (f g:In->Set),
  (forall i, sub (f i) (g i)) -> sub (uniont f) (uniont g).
Lemma intersection_monotone: forall (In:Set) (f g:In->Set), nonemptyT In ->
  (forall i, sub (f i) (g i)) -> sub (intersectiont f) (intersectiont g).
Lemma union_monotone2: forall a b f,
  sub a b -> sub (unionf a f) (unionf b f).
Lemma intersection_monotone2: forall a b f,
  sub a b -> nonempty a -> sub (intersectionf b f) (intersectionf a f).
```

Proposition 2 [2, p. 93] states associativity of union and intersection. It says:

$$\bigcup_{\iota\in I} X_\iota = \bigcup_{\lambda\in L}\left(\bigcup_{\iota\in J_\lambda} X_\iota\right), \qquad \bigcap_{\iota\in I} X_\iota = \bigcap_{\lambda\in L}\left(\bigcap_{\iota\in J_\lambda} X_\iota\right).$$

```
Theorem union_assoc: forall sf sg f g,
  sf = unionf sg g ->
  unionf sf f = unionf sg (fun l => unionf (g l) f).
Theorem intersection_assoc: forall sf sg f g,
  nonempty sg ->  (forall i, inc i sg -> nonempty (g i)) ->
  sf = unionf sg g ->
  intersectionf sf f = intersectionf sg (fun l => intersectionf (g l) f).
```

Proposition 3 [2, p. 94] says that if Γ is a correspondence, $\Gamma\langle\bigcup X_\iota\rangle = \bigcup\Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap\Gamma\langle X_\iota\rangle$. Proposition 4 [2, p. 95] says that we have equality if Γ is the inverse of a function, and,

as a consequence, if Γ is an injective function. In fact, we use the canonical decomposition $\Gamma = i \circ g$, where g is the restriction of Γ on its image (hence is bijective), and i is the inclusion map from the image of Γ to its target (see lemma *canonical_decomposition1*). Then $\Gamma\langle x\rangle = g^{-1}\langle x\rangle$ for every set x.

```
Theorem image_of_union: forall (In:Set) (f:In->Set) g,
  is_correspondence g ->
  image_by_fun g (uniont f) =
  uniont (fun i => image_by_fun g (f i)).
Theorem image_of_intersection: forall (In:Set) (f:In->Set) g,
  is_correspondence g -> nonemptyT In->
  sub (image_by_fun g (intersectiont f))
  (intersectiont (fun i => image_by_fun g (f i))).
Theorem inv_image_of_intersection: forall (In:Set) (f:In->Set) g,
  is_function g ->  nonemptyT In ->
  (inv_image_by_fun g (intersectiont f)) =
  (intersectiont (fun i => inv_image_by_fun g (f i))).
Lemma inj_image_of_intersection: forall (In:Set) (f:In->Set) g,  (* 28 *)
  injective g -> nonemptyT In ->
  (image_by_fun g (intersectiont f))
  = (intersectiont (fun i => image_by_fun g (f i))).
```

5.3 Complements of unions and intersections

Assume $X_\iota \subset X$, $Y_\iota = X \setminus X_\iota$. Then the intersection (resp. union) of the X_ι is the union (resp. intersection) of the Y_ι as subsets of X. This is Proposition 5 [2, p. 96].

```
Theorem complementary_union: forall (In:Set) (f:In-> Set) x,
  nonemptyT In ->  (forall i, sub (f i) x) ->
  complement x (uniont f) = intersectiont (fun i=> complement x (f i)).
Theorem complementary_intersection: forall (In:Set) (f:In-> Set) x,
  nonemptyT In ->   (forall i, sub (f i) x) ->
  complement x (intersectiont f) = uniont (fun i=> complement x (f i)).
Lemma complementary_union1: forall sf f x,
  nonempty sf ->  (forall i, inc i sf -> sub (f i) x) ->
  complement x (unionf sf f) = intersectionf sf (fun i=> complement x (f i)).
Lemma complementary_intersection1:  forall sf f x,
  nonempty sf ->  (forall i, inc i sf -> sub (f i) x) ->
  complement x (intersectionf sf f) = unionf sf (fun i=> complement x (f i)).
```

5.4 Union and intersection of two sets

Bourbaki defines the union and intersection of two sets A and B as the union and intersection of the identity function on the doubleton $\{A, B\}$. This was defined as *union2* and *intersection2*. All results shown here are easy. Some formulas are implemented in the file *set1.v*.

```
Lemma union_of_twosets_aux: forall x y f,
  unionf(doubleton x y) f = union2 (f x) (f y).
Lemma intersection_of_twosets_aux: forall x y f,
  intersectionf(doubleton x y) f = intersection2 (f x) (f y).
```

```
Lemma union_of_twosets: forall x y,
  unionf(doubleton x y)(fun w => w) = union2 x y.
Lemma intersection_of_twosets: forall x y,
  intersectionf(doubleton x y)(fun w => w) = intersection2 x y.
Lemma union_doubleton: forall x y,
  union2 (singleton x)(singleton y) = doubleton x y.
```

We have:

$$\{x\} \cup \{y\} = \{x, y\}, \qquad x \cup x = x, \qquad x \cap x = x, \qquad x \cap y = y \cap x, \qquad x \cup y = y \cup x.$$

We have:

$$x \cup (y \cup z) = (x \cup y) \cup z, \qquad x \cap (y \cap z) = (x \cap y) \cap z,$$
$$x \cup (y \cap z) = (x \cup y) \cap (x \cup z), \qquad x \cap (y \cup z) = (x \cap y) \cup (x \cap z).$$

```
Lemma union2assoc:  forall x y z,
  union2 x (union2 y z) = union2 (union2 x y) z.
Lemma intersection2assoc:  forall x y z,
  intersection2 x (intersection2 y z) = intersection2 (intersection2 x y) z.
Lemma intersection_union_distrib1:  forall x y z,
  union2 x (intersection2 y z) = intersection2 (union2 x y) (union2 x z).
Lemma intersection_union_distrib2:  forall x y z,
  intersection2 x (union2 y z) = union2 (intersection2 x y) (intersection2 x z).
```

We have $x \subset y$ if and only if $x \cup y = y$. We have $x \subset y$ if and only if $x \cap y = x$. We have:

$$z \setminus (x \cup y) = (z \setminus x) \cap (z \setminus y), \qquad z \setminus (x \cap y) = (z \setminus x) \cup (z \setminus y).$$

```
Lemma union2_comp: forall x y z,
  complement z (union2 x y) = intersection2 (complement z x)(complement z y).
Lemma intersection2_comp: forall x y z,
  sub x z -> sub y z ->
  complement z (intersection2 x y) = union2 (complement z x)(complement z y).
```

We have $x \cup (z \setminus x) = z$ and $x \cap (z \setminus x) = \emptyset$. If g is a correspondence, we have $g\langle x \cup y\rangle = g\langle x\rangle \cup g\langle y\rangle$ and $g\langle x \cap y\rangle \subset g\langle x\rangle \cap g\langle y\rangle$. Equality holds if g is an injective function or $g = f^{-1}$ where f is a function.

```
Lemma union2_complement: forall x z,
  sub x z -> union2 x (complement z x) = z.
Lemma intersection2_complement: forall x z,
  sub x z -> intersection2 x (complement z x) = emptyset.
Lemma image_of_union2: forall g x y,
  is_correspondence g ->
  image_by_fun g (union2 x y) = union2 (image_by_fun g x)(image_by_fun g y).
Lemma image_of_intersection2: forall g x y,
  is_correspondence g ->
  sub (image_by_fun g (intersection2 x y))
  (intersection2 (image_by_fun g x) (image_by_fun g y)).
Lemma inv_image_of_intersection2: forall g x y,
  is_function g ->
  inv_image_by_fun g (intersection2 x y) =
  intersection2 (inv_image_by_fun g x)(inv_image_by_fun g y).
Lemma inj_image_of_intersection2: forall g x y,
  injective g ->
  image_by_fun g (intersection2 x y)
  = intersection2 (image_by_fun g x)(image_by_fun g y).
```

If f is a function from A into B, then we have $f^{-1}\langle B \setminus x\rangle = A \setminus f^{-1}\langle x\rangle$ and $f\langle A \setminus x\rangle = B \setminus f\langle x\rangle$ if f is a injective with range B (Proposition 6, [2, p. 98]).

```
Lemma inv_image_of_comp: forall f x,
  is_function f -> sub x (target f)  ->
  inv_image_by_fun f (complement (target f) x) =
  complement (inv_image_by_fun f (target f))(inv_image_by_fun f x).
Lemma inj_image_of_comp: forall f x,
  injective f -> sub x (source f)  ->
  image_by_fun f (complement (source f) x) =
  complement (image_by_fun f (source f))(image_by_fun f x).
```

5.5 Coverings

A *covering* of a set X is a family X_ι whose union contains X. We give two definitions, in the first case, the family is defined by a function, and in the second one, by the graph of a function. We show that these definitions agree.

```
Definition covering f x :=  fgraph f & sub x (unionb f).
Definition covering_f sf f x :=  sub x (unionf sf f).
Definition covering_s f x :=  sub x (union f).
Lemma covering_pr: forall f x,
   fgraph f -> covering f x = covering_s (range f) x.
Lemma covering_f_pr: forall sf f x,
  covering_f sf f x = covering_s (range (L sf f)) x.
```

We say that a covering $(Y_\kappa)_{\kappa\in K}$ refines $(X_\iota)_{\iota\in I}$ if for all κ there is ι such that $Y_\kappa \subset X_\iota$. This definition will be extended to set coverings: the definition *coarser_c y y'* says that the set of sets y' refines y. In other words, for all $a \in y'$ there is $b \in y$ such that $a \subset b$. This can be also be read as: y' is finer than y as as y is coarser then y'. We will show that this is an order on the set of all partitions; we show here transitivity.

```
Definition coarser_covering sf f sg g :=
  forall j, inc j sg -> exists i, inc i sf & sub (g j) (f i).

Definition coarser_c y y' :=
  forall a, inc a y' -> exists b, inc b y & sub a b.

Lemma coarser_same: forall sf f sg g ,
  coarser_covering sf f sg g = coarser_c (range (L sf f))(range (L sg g)).
Lemma coarser_transitive : forall y y' y'',
  coarser_c y y' -> coarser_c y'  y'' -> coarser_c y y''.

Lemma sub_covering: forall Ia Ib f x,
  covering_f Ia f x ->  covering_f Ib f x -> sub Ib Ia ->
  coarser_covering Ia f Ib f.
```

Given two families X_ι and Y_κ, we can consider the family $X_\iota \cap Y_\kappa$. Given two sets of sets X and Y, we can consider the set of elements of the form $a \cap b$ for $a \in X$ and $b \in Y$. Hence, given two coverings X_ι and Y_κ of Z we find a covering $i(X_\iota, Y_\kappa)$ of Z that refines X_ι and Y_κ, this is the sup for the coarser ordering (ordering are defined in the second part of this report).

```
Definition intersection_covering f g :=
```

```
  fun z => intersection2 (f (P z)) (g (Q z)).

Definition intersection_covering2 x y:=
  range(L (product x y)(intersection_covering (fun w => w) (fun w => w))).
Lemma intersection_covering2_pr: forall x y z,
  inc z (intersection_covering2 x y) =
  exists a, exists b, inc a x & inc b y & intersection2 a b = z.
Lemma product_is_covering: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  covering_f (product sf sg) (intersection_covering f g) x.
Lemma intersection_covering_coarser1: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  coarser_covering sf f (product sf sg) (intersection_covering f g).
Lemma intersection_covering_coarser2: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  coarser_covering sg g (product sf sg) (intersection_covering f g).
Lemma intersection_covering_coarser3: forall sf f sg g  sh h x,
  covering_f sf f x -> covering_f sg g  x -> covering_f sh h  x ->
  coarser_covering sf f sh h ->
  coarser_covering sg g  sh h ->
  coarser_covering (product sf sg) (intersection_covering f g) sh h.
```

We show here the equivalent properties for sets of sets. Essentially, we prove that $i(x,y)$ is the least upper bound for the order defined by *coarser_c* (which is defined on the set of partitions, as will be seen later).

```
Lemma product_is_covering2: forall u v x,
  covering_s u x -> covering_s v  x ->
  covering_s (intersection_covering2 u v) x.
Lemma intersection_cov_coarser1: forall f g x,
  covering_s f x -> covering_s g x ->
  coarser_c f (intersection_covering2 f g).
Lemma intersection_cov_coarser2: forall f g x,
  covering_s f x -> covering_s g x ->
  coarser_c g (intersection_covering2 f g).
Lemma intersection_cov_coarser3: forall f g h x,
  covering_s f x -> covering_s g x -> covering_s h x ->
  coarser_c f h ->  coarser_c g h ->
  coarser_c (intersection_covering2 f g) h.
```

If X_ι is a covering and g a function, then the family of sets $g^{-1}\langle X_\iota\rangle$ is a covering; if g is surjective, then $g\langle X_\iota\rangle$ is a covering.

```
Lemma image_of_covering: forall sf f g,
  surjective g -> covering_f sf f (source g)
  -> covering_f sf (fun w => image_by_fun g (f w)) (target g).
Lemma inv_image_of_covering: forall sf f g,
  is_function g -> covering_f sf f (target g)
  -> covering_f sf (fun w => inv_image_by_fun g (f w)) (source g).
Lemma product_of_covering: forall sf f sg g x y,
  covering_f sf f x -> covering_f sg g y ->
  covering_f (product sf sg) (fun z => product (f (P z)) (g (Q z)))
  (product x y).
```

Proposition 7 [2, p. 99] says that if X_ι is a covering of E, then two functions that agree on each X_ι agree on E, and a function defined on each X_ι can be extended to E if the obvious compatibility conditions hold.

We modified the theorem, by adding the condition that the graph is the union of the graphs, and the range is the union of the ranges. Initial size of the proof was 42 lines; it is now 35 (without the assumption on the graph, uniqueness is not trivial).

```
Definition function_prop f s t:=
  is_function f & source f = s & target f = t.
Definition function_prop_sub f s t:=
  is_function f & source f = s & sub (target f) t.

Lemma agrees_on_covering: forall sf f x g g',
  covering_f sf f x -> is_function g -> is_function g' ->
  source g = x -> source g' = x ->
  (forall i, inc i sf ->  agrees_on (intersection2 x (f i)) g g') ->
  agrees_on x g g'.

Lemma prolongation_covering1: forall sf f t h,
  (forall i, inc i sf -> function_prop (h i)(f i) t ) ->
  (forall i j, inc i sf -> inc j sf ->
    agrees_on (intersection2 (f i) (f j)) (h i) (h j)) ->
  exists g, function_prop g (unionf sf f) t &
    graph g = (unionf sf (fun i => (graph (h i)))) &
    range(graph g) = (unionf sf (fun i => (range (graph (h i))))) &
    (forall i, inc i sf -> agrees_on (f i) g (h i)).  (* 34 *)

Lemma prolongation_covering: forall sf f t h,
  (forall i, inc i sf -> function_prop (h i) (f i) t) ->
  (forall i j, inc i sf -> inc j sf ->
    agrees_on (intersection2 (f i) (f j)) (h i) (h j)) ->
  exists_unique (fun g => function_prop g (unionf sf f) t &
    (forall i, inc i sf -> agrees_on (f i) g (h i))).
```

5.6 Partitions

Definition 7 in Bourbaki [2, p. 100] is: a *partition* of a set E is a family of *non-empty* mutually disjoints subsets of E which covers E; the phrase non-empty is missing in the French version. Here *partition_fam* is a family X_ι of mutually disjoint sets, whose union is E; *partition_s* is a set of sets with this property, and a *partition* is a set of non-empty sets.

```
Definition disjoint x y := intersection2 x y = emptyset.
Definition mutually_disjoint f :=
  (forall i j, inc i (domain f) -> inc j (domain f) ->
    i = j \/ (disjoint (V i f) (V j f))).
Definition partition_s y x:=
  (union y = x) &
  (forall a b, inc a y -> inc b y -> a = b \/ disjoint a b).
Definition partition y x:=
  (union y = x) &
  (forall a, inc a y -> nonempty a) &
  (forall a b, inc a y -> inc b y -> a = b \/ disjoint a b).
Definition partition_fam f x:=
  fgraph f & mutually_disjoint f & unionb f = x.
```

We list below some properties of partitions. The important property if that if (X_ι) is a partition of E, each element of E is in a unique X_ι.

```
Lemma partitionset_pr: forall y x,
  partition y x -> partition_s y x.
Lemma partition_same: forall y x,
  partition_s y x -> partition_fam (identity y) x.
Lemma partition_same2: forall y x,
  partition_fam y x -> partition_s (range y) x.
Lemma partitions_is_covering: forall y x,
  partition_s y x -> covering_s y x.
Lemma partition_fam_is_covering: forall y x,
  partition_fam y x -> covering y x.
Lemma partition_inc_exists: forall f x y,
  partition_fam f x -> inc y x -> exists i, (inc i (domain f) & inc y (V i f)).
Lemma partition_inc_unique: forall f x i j y,
  partition_fam f x -> inc i (domain f) -> inc y (V i f) ->
  inc j (domain f) -> inc y (V j f) -> i = j.
```

We construct here a function that maps a to x and b to y. This function is well-defined if $a = x$ and $b = y$, since it is the identity on the doubleton $\{x, y\}$. It is also well defined if a and b are distinct elements. We will use the fact that *two_points* is a set with exactly two elements *TPa* and *TPb*.

```
Definition variant a x y := (fun z:Set => Yo (z = a) x y).
Definition Lvariant a b x y := L (doubleton a b) (variant a x y).
Definition Lvariantc f g:=  Lvariant TPa TPb f g.

Definition partition_with_complement x j :=
  Lvariantc j (complement x j).

Lemma variant_if_rw: forall a x y z,
  z = a -> variant a x y z = x.
Lemma variant_if_not_rw: forall a x y z,
  z <> a -> variant a x y z = y.
Lemma V_variant_a : forall a b x y,
  V a (Lvariant a b x y)  = x.
Lemma V_variant_b : forall a b x y,
  b <> a ->  V b (Lvariant a b x y)  = y.
Lemma axioms_variant : forall a b x y,
  fgraph (Lvariant a b x y).
Lemma domain_variant : forall a b x y,
  domain (Lvariant a b x y) = doubleton a b.

Lemma axioms_Lvariantc : forall x y,
  fgraph (Lvariantc x y).
Lemma domain_Lvariantc: forall f g,
  domain (Lvariantc f g) = two_points.
Lemma nonempty_domain_Lvariantc: forall f g,
  nonempty (domain (Lvariantc f g)).
Lemma V_variant_ca : forall x y,  V TPa (Lvariantc x y) = x.
Lemma V_variant_cb : forall x y,  V TPb (Lvariantc x y) = y.
```

If X is a subset of E then X and E \ X form a partition of X (it is a non-empty partition only if X is neither empty nor E).

```
Lemma disjoint_pr: forall a b,
  (forall u, inc u a -> inc u b -> False) -> disjoint a b.
```

```
Lemma disjoint_complement : forall x y,
  disjoint y (complement x y).
Lemma disjoint_symmetric : forall x y,
  disjoint x y -> disjoint y x.
Lemma is_partition_with_complement: forall x j,
  sub j x -> partition_fam (partition_with_complement x j) x.
```

The set of non-empty partitions on X can be ordered by the finer ordering on coverings; we give here the smallest and largest element of the set. If X_ι is a partition family, then the mapping $\iota \mapsto X_\iota$ is injective (we use the fact that X_ι is not empty). Inverse images of disjoint sets by a function are disjoint.

```
Definition largest_partition x := range(L x (fun z => singleton z)).
Definition smallest_partition x :=   (singleton x).
Lemma partition_smallest: forall x,
   nonempty x -> partition_set (smallest_partition x) x.
Lemma largest_partition_pr: forall x z,
  inc z (largest_partition x) = exists w, inc w x & singleton w = z.
Lemma partition_largest: forall x, partition_set (largest_partition x) x.
Definition injective_graph f:=
  fgraph f &
  (forall x y, inc x (domain f) -> inc y (domain f) ->
    V x f = V y f -> x = y).

Lemma injective_partition: forall f x,
  partition_fam f x -> (forall i,  inc i (domain f) -> nonempty (V i f))
  -> injective_graph f.
Lemma partition_fam_partition: forall f x,
  partition_fam f x -> (forall i,  inc i (domain f) -> nonempty (V i f))
  -> partition(range f) x.
Lemma inv_image_disjoint :  forall g x y,
  is_function g ->  disjoint x y ->
  disjoint (inv_image_by_fun g x)(inv_image_by_fun g y).
```

Proposition 8 [2, p. 100] is an immediate consequence of Proposition 7. If $(X_\iota)_\iota$ is a partition of X and $f_\iota \in \mathscr{F}(X_\iota, T)$, then there exists a unique $f \in \mathscr{F}(X, T)$ that extends every f_ι. The assumption is that f_ι is a function defined on X_ι, with target T. We give a variant (without uniqueness) where the target of f_ι is a subset of T. The set of function $\mathscr{F}$ will be defined later.

```
Theorem prolongation_partition: forall f x t h,
  partition_fam f x ->
  (forall i, inc i (domain f) -> function_prop (h i) (V i f) t) ->
  exists_unique (fun g => function_prop g x t &
    (forall i, inc i (domain f) -> agrees_on (V i f) g (h i))).

Theorem prolongation_partition: forall f x t h,
  partition_fam f x ->
  (forall i, inc i (domain f) -> is_function (h i)) ->
  (forall i, inc i (domain f) -> target (h i) = t) ->
  (forall i, inc i (domain f) -> source (h i) = (V i f)) ->
  exists_unique (fun g => is_function g & source g = x &
    target g = t & (forall i, inc i (domain f) -> agrees_on (V i f) g (h i))).
```

5.7 Sum of a family of sets

Proposition 9 [2, p. 100] says that, for any family X_ι, there exists a family X'_ι of sets equipotent to X_ι, that are mutually disjoint, and a set X that is the union of these sets. After that we define the *sum* of a family as the union of the family $X_\iota \times \{\iota\}$. These sets form a partition of the sum. Proposition 10 [2, p. 101] says that that if X_ι is a family with union A and sum S, there is a bijection between A and S if the family is disjoint. A comment says that there always exists a surjection.

```
Theorem disjoint_union_lemma : forall f,
  fgraph f -> exists g, exists x,
    fgraph g &  x = unionb g &
    (forall i, inc i (domain f) -> equipotent (V i f) (V i g))
    & mutually_disjoint g.   (* 53 *)

Definition disjoint_union_fam f :=
  (L (domain f)(fun i => product (V i f) (singleton i))).

Definition disjoint_union f :=
  unionb (disjoint_union_fam f).

Lemma disjoint_union_disjoint:
  forall f, fgraph f ->
    mutually_disjoint(disjoint_union_fam f).

Lemma partion_union_disjoint:  forall f, fgraph f ->
  partition_fam (disjoint_union_fam f)  (disjoint_union f).
Theorem disjoint_union_pr: forall f,
  fgraph f -> exists x,
    source x = disjoint_union f &
    target x = unionb f &
    surjective x &
    (mutually_disjoint f -> bijective x).   (* 43 *)
```

Chapter 6

Product of a family of sets

6.1 The axiom of the set of subsets

Bourbaki has an axiom (Axiom 4 in the English edition) that asserts the existence, for every set X, of the set of subsets of X. This is sometimes called the powerset of X, it is denoted by $\mathfrak{P}(X)$. C. Simpson has defined it in Section 2.8. We state here two easy lemmas.

```
Lemma powerset_monotone: forall a b,
  sub a b -> sub (powerset a) (powerset b).
Lemma powerset_emptyset :
  powerset emptyset = singleton emptyset.
```

If f is a correspondence from A to B, then $f\langle X\rangle \subset B$ whenever $X \subset A$. This gives a function from $\mathfrak{P}(A)$ to $\mathfrak{P}(B)$, called *extension to sets of subsets*. If we denote it by $\hat{f}$, then the extension of $f \circ g$ is $\hat{f} \circ \hat{g}$. The extension of the identity is the identity. The extension of an inverse is an inverse of the extension; this can be more formally restated in Proposition 1 [2, p. 101] as: if f is surjective (resp. injective), then its restriction to the set of sets is surjective (resp. injective).

```
Definition extension_to_parts f :=
  BL(image_by_fun f) (powerset (source f)) (powerset (target f)).
Lemma axioms_etp: forall f,  is_correspondence f ->
  transf_axioms (image_by_fun f) (powerset (source f))
  (powerset (target f)).
Lemma function_etp: forall f,
  is_correspondence f -> is_function  (extension_to_parts f).
Lemma W_etp: forall f x,
  is_correspondence f -> sub x (source f)
  -> W x (extension_to_parts f) = image_by_fun f x.
Lemma composable_etp: forall f g,
  composableC g f ->
  composable (extension_to_parts g)  (extension_to_parts f).
Lemma compose_etp: forall f g,
  composableC g f ->
  compose (extension_to_parts g)  (extension_to_parts f)
  = extension_to_parts (compose g f).
Lemma identity_etp: forall x,
  extension_to_parts (identity_fun x) = identity_fun (powerset x).
Lemma composable_for_function: forall f g, composable g f -> composableC g f.
```

```
Theorem surjective_etp: forall f,
  surjective f -> surjective (extension_to_parts f).
Theorem injective_etp: forall f,
  injective f -> injective (extension_to_parts f).
```

6.2 Set of mappings of one set into another

The set of all graphs of functions from E to F is denoted by F^E: this is a subset of the powerset of $E \times F$. The set of all functions, namely the set of triples (G, E, F) where $G \in F^E$, is denoted by $\mathscr{F}(E;F)$. In the previous chapter, we have defined the set of graphs of correspondences. There is no set containing all f satisfying the property *is_function f*, so that we must write something like $g \in \mathscr{F}(E;F)$ if and only if there is a function f associated to g. A bijection from E to itself is called a *permutation* of E.

```
Definition set_of_functions x y :=
  Zo (set_of_correspondences x y)(fun z => fgraph (P z)& x = domain (P z)).
Definition set_of_permutations E :=
  Zo (set_of_functions E E)(fun z=> bijective(inv_corr_value z)).

Lemma inc_set_of_functions: forall f,
  is_function f -> inc(corr_value f) (set_of_functions (source f) (target f)).
Lemma set_of_functions_inc: forall x y z,
  inc z (set_of_functions x y) -> exists f,
    is_function  f &  source f = x & target f = y & corr_value f=z.
Lemma inc_set_of_functionsb: forall x y f,
  transf_axioms f x y -> inc (corr_value (BL f x y)) (set_of_functions x y)
Lemma inc_set_of_functionsc: forall f x y,
  is_function f ->  x = source f -> y = target f ->
  inc (corr_value f) (set_of_functions x y).
Lemma sof_value_pr:  forall x y z,
  inc z (set_of_functions x y) ->
    (is_function (sof_value x y z) &
    source (sof_value x y z) = x &
    target (sof_value x y z) = y &
    corr_value (sof_value x y z) =z).   (* 14 *)

Lemma inc_set_of_functionsa: forall (x y:Set) (f:x->y),
  inc (corr_value (acreate f)) (set_of_functions x y).
Lemma set_of_functionsC_inc: forall x y z,
  inc z (set_of_functions x y) -> exists f:x->y, corr_value (acreate f) =z.
Lemma set_of_permutationsC_inc: forall x z,
  inc z (set_of_permutations x) -> exists f:x->x,
    bijectiveC f & corr_value (acreate f) =z.
Lemma inc_set_of_permutationsC: forall x (f:x->x),
  bijectiveC f -> inc (corr_value (acreate f)) (set_of_permutations x).
```

We introduce now F^E. It is canonically isomorphic to $\mathscr{F}(E;F)$; this means that using one set or the other does not change the size of a proof.

```
Definition set_of_gfunctions x y :=
  Zo (powerset (product x y))(fun z => fgraph z & x = domain z).
Lemma inc_set_of_gfunctions: forall f,
  is_function f -> inc(graph f) (set_of_gfunctions (source f) (target f)).
Lemma inc_set_of_gfunctionsa: forall x y (f:x->y),
```

```
  inc (graph (acreate f)) (set_of_gfunctions x y).
Lemma set_of_gfunctions_inc: forall x y z,
  inc z (set_of_gfunctions x y) -> exists f,
    is_function  f &  source f = x & target f = y & graph f =z.
```

The set of partial functions from x to y will be used later on. It is the union of the sets of functions from x' to y, where $x' \subset x$. We give two properties of this set.

```
Definition set_of_sub_functions x y:=
  unionf(powerset x)(fun z=> (set_of_functions z y)).

Lemma set_of_sub_functions_pr: forall x y z,
  inc z (set_of_sub_functions x y) =
  exists f, is_function  f &  sub (source f) x & target f = y & corr_value f=z.
Lemma set_of_sub_functions_pr1: forall x y z,
  let f :=  inv_corr_value z in
  inc z (set_of_sub_functions x y) =
  (is_function  f &  sub (source f) x & target f = y & corr_value f=z).
```

We list below some properties of exceptional functions. Remember that a small set has at most one element. We then show that there is an obvious bijection between $\mathscr{F}(\mathrm{E};\mathrm{F})$ and F^{E}.

```
Lemma empty_source_graph: forall f,
  is_function f -> source f = emptyset -> graph f = emptyset.
Lemma empty_target_graph: forall f,
  is_function f -> target f = emptyset -> graph f = emptyset.

Lemma small_set_of_functions_source: forall x y,
  x = emptyset -> small_set (set_of_functions x y).
Lemma small_set_of_functions_target: forall x y,
  y = emptyset -> small_set (set_of_functions x y).
Lemma empty_set_of_functions_target: forall x y,
  (x = emptyset \/ nonempty y) -> nonempty (set_of_functions x y).
Lemma set_of_functions_extens: forall x y a b,
  inc a (set_of_functions x y) -> inc b (set_of_functions x y) ->
  P a = P b -> a = b.
Lemma axioms_graph_function: forall x y,
  transf_axioms P (set_of_functions x y)  (set_of_gfunctions x y).
Lemma bijective_graph_function: forall x y,
  bijective (BL P (set_of_functions x y) (set_of_gfunctions x y)).
```

$$\begin{array}{ccc} \mathrm{E} & \xrightarrow{f} & \mathrm{F} \\ {\scriptstyle u}\uparrow & & \downarrow{\scriptstyle v} \\ \mathrm{E}' & \xrightarrow[f']{} & \mathrm{F}' \end{array} \qquad \text{(compose3function)}$$

Given $f \in \mathscr{F}(\mathrm{E};\mathrm{F})$, we construct $f' \in \mathscr{F}(\mathrm{E}';\mathrm{F}')$ via $f' = v \circ f \circ u$, provided that u is a function from E' to E and v is a function from F into F'. The trouble with this definition is that an element of $\mathscr{F}(\mathrm{E};\mathrm{F})$ is not a function, but a triple, and we do not have a definition of compositions for such objects. We first convert f into a function, then take the triple associated to f'. Proposition 2 [2, p. 102] says that if u is surjective and v is injective, then this mapping is injective; if u is injective and v is surjective, then this mapping is surjective. The situation is a bit more tricky when some sets are empty.

```
Definition compose3function u v :=
  BL (fun f => corr_value (compose (compose v
          (sof_value (target u) (source v) f)) u))
  (set_of_functions (target u) (source v))
  (set_of_functions (source u) (target v)).

Lemma axioms_c3f: forall u v,
  is_function u -> is_function v ->
  transf_axioms (fun f => corr_value (compose (compose v
          (sof_value (target u) (source v) f)) u))
  (set_of_functions (target u) (source v))
  (set_of_functions (source u) (target v)).
Lemma function_c3f: forall u v,
  is_function u -> is_function v -> is_function(compose3function u v).
Lemma W_c3f: forall u v f,
  is_function u -> is_function v ->
  is_function f -> source f = target u -> target f = source v ->
  W (corr_value f) (compose3function u v) = corr_value (compose (compose v f)u).
Theorem injective_c3f: forall u v,
  surjective u -> injective v -> injective  (compose3function u v).   (* 22 *)
Theorem surjective_c3f: forall u v,
  (nonempty (source u) \/ (nonempty (source v)) \/ (nonempty (target v))
    \/ target u = emptyset) ->
  injective u -> surjective v -> surjective  (compose3function u v).    (* 31 *)
Lemma bijective_c3f: forall u v,
  bijective u -> bijective v -> bijective  (compose3function u v).
```

We now define the canonical bijections from $\mathscr{F}(B\times C;A)$ into $\mathscr{F}(B;\mathscr{F}(C;A)$ or $\mathscr{F}(C;\mathscr{F}(B;A))$. For any function $f(x,y)$ we can fix one of the variables to get a function.

```
Definition first_partial_fun f y:=
  BL(fun x => W (J x y) f) (domain (source f)) (target f).
Definition second_partial_fun f x:=
  BL(fun y => W (J x y) f) (range (source f)) (target f).
Definition first_partial_function f:=
  BL(fun y => corr_value (first_partial_fun f y)) (range (source f))
  (set_of_functions (domain (source f)) (target f)).
Definition second_partial_function f:=
  BL(fun x => corr_value(second_partial_fun f x)) (domain (source f))
  (set_of_functions (range (source f)) (target f)).
Definition first_partial_map b c a:=
  BL (fun f=> corr_value (first_partial_function (sof_value (product b c) a f)))
  (set_of_functions (product b c) a)
  (set_of_functions c (set_of_functions b a)).
Definition second_partial_map b c a:=
  BL (fun f=> corr_value(second_partial_function (sof_value (product b c) a f)))
  (set_of_functions (product b c) a)
  (set_of_functions b (set_of_functions c a)).
```

The next lemmas show that for fixed x, the partial application f_x that maps y to $f(x,y)$ is a function. Similarly for f_y.

```
Lemma axioms_fpf: forall f y,
  partial_fun_axioms f -> inc y (range (source f)) ->
  transf_axioms (fun x => W (J x y) f) (domain (source f)) (target f).
Lemma axioms_spf: forall f x,
```

```
  partial_fun_axioms f -> inc x (domain (source f)) ->
  transf_axioms (fun y => W (J x y) f) (range (source f)) (target f).
Lemma function_fpf :forall f y,
  partial_fun_axioms f -> inc y (range (source f)) ->
  is_function (first_partial_fun f y).
Lemma function_spf :forall f x,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  is_function (second_partial_fun f x).
Lemma W_fpf :forall f x y,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  W x (first_partial_fun f y) = W (J x y) f.
Lemma W_spf :forall f x y,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  W y (second_partial_fun f x) = W (J x y) f.
```

The next lemmas show that both $x \mapsto f_x$ and $y \mapsto f_y$ are functions.

```
Lemma axioms_fpfa: forall f,
  partial_fun_axioms f ->
  transf_axioms (fun y => corr_value (first_partial_fun f y))(range (source f))
  (set_of_functions (domain (source f)) (target f)).
Lemma axioms_spfa: forall f ,
  partial_fun_axioms f ->
  transf_axioms (fun x => corr_value(second_partial_fun f x))(domain (source f))
  (set_of_functions (range (source f)) (target f)).
Lemma function_fpfa: forall f,
  partial_fun_axioms f -> is_function (first_partial_function f).
Lemma function_spfa: forall f ,
 partial_fun_axioms f -> is_function (second_partial_function f).
Lemma W_fpfa: forall f y,
  partial_fun_axioms f -> inc y  (range (source f)) ->
  W y (first_partial_function f) = corr_value(first_partial_fun f y).
Lemma W_spfa: forall f x,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  W x (second_partial_function f) = corr_value(second_partial_fun f x).
```

Denote the mapping $x \mapsto f_x$ by $\tilde{f}$. We show here that the mapping $f \mapsto \tilde{f}$ is a function. We assume that the source is nonempty.

```
Lemma axioms_fpfb: forall a b c,
  nonempty b -> nonempty c ->
  transf_axioms (fun f=>
    corr_value (first_partial_function (sof_value (product b c) a f)))
  (set_of_functions (product b c) a)
  (set_of_functions c (set_of_functions b a)).
Lemma axioms_spfb: forall a b c,
  nonempty b -> nonempty c ->
  transf_axioms (fun f=>
    corr_value(second_partial_function (sof_value (product b c) a f)))
  (set_of_functions (product b c) a)
  (set_of_functions b (set_of_functions c a)).
Lemma W_fpfb: forall  a b c f,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  W f (first_partial_map a b c) =
```

```
  corr_value(first_partial_function (sof_value (product a b) c f)).
Lemma W_spfb: forall a b c f,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  W f (second_partial_map a b c) =
  corr_value(second_partial_function (sof_value (product a b) c f)).
Lemma WW_fpfb: forall a b c f x,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  inc x (product a b) ->
  W (P x) (sof_value a c (W (Q x)
    (sof_value b (set_of_functions a c) (W f (first_partial_map a b c))))) =
  W x (sof_value (product a b) c f).     (* 15 *)
Lemma WW_spfb: forall a b c f x,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  inc x (product a b) ->
  W (Q x) (sof_value b c (W (P x)
    (sof_value a (set_of_functions b c) (W f (second_partial_map a b c))))) =
  W x (sof_value (product a b) c f).      (* 15 *)
```

We now prove the main result, Proposition 3 of [2, p. 103].

```
Theorem bijective_fpfa: forall a b c,
  nonempty a -> nonempty b -> bijective (first_partial_map a b c).     (* 48 *)
Theorem bijective_spfa: forall a b c,
  nonempty a -> nonempty b -> bijective (second_partial_map a b c).    (* 48 *)
```

6.3 Definition of the product of a family of sets

An element of the product of two sets X_1 and X_2 is a pair of elements of X_1 and X_2, an element of the product of n sets $X_1,\dots,X_n$ is a tuple $(x_1,\dots,x_n)$, and thus an element of the product of a family $(X_\iota)_{\iota\in I}$ is a family $(x_\iota)_{\iota\in I}$ such that $x_\iota \in X_\iota$. We give three definitions of the product, in the same way as we gave three definitions for the union or the intersection. Note that the family (X_ι) can be defined by a function $f : a \to E$, but the family (x_ι) cannot, because there is not set containing such objects; an element of a product is the graph of a function. We define below *gbcreate*, a variant of *gacreate*, that is the graph of *acreate*, that converts $f : a \to E$ into a graph.

```
Definition gbcreate (a:Set) (f:a->Set) := (IM (fun y:a => J (Ro y) (f y))).
Lemma inc_gbcreate: forall (a:Set) (f:a->E) x,
  inc x (gbcreate f) = exists y:a, J (Ro y) (f y) =x.
Lemma graph_gbcreate: forall (a:Set) (f:a-> Set), is_graph (gbcreate f).
Lemma fgraph_gbcreate: forall (a:Set) (f:a-> Set), fgraph (gbcreate f).
Lemma domain_gbcreate : forall (a:Set) (f:a-> Bsezt),   domain (gbcreate f) = a.
Lemma V_gbcreate : forall (a:Set) (f:a-> Set) (x:a),  V (Ro x) (gbcreate f) = f x.
```

Given a family $(X_\iota)_{\iota\in I}$ of sets defined on I, we may consider functions f such that $f(\iota) \in X_\iota$. The target of $f(\iota)$ is in the union $A = \bigcup X_\iota$ and the graph is an element of $\mathfrak{P}(I \times A)$. Thus, we define the product $\prod X_\iota$ as the set of all elements $\mathfrak{P}(I \times A)$ that are are graphs of functions with this property. If all X_ι are the same set E, then $A = E$, this justifies the notation E^I for the set of functional graphs from I to E since it is the product of a constant family.

In the basic definition, the family is defined by the graph of a function, but in some cases, it is easier to consider a function defined on the type I. If I has two elements, the product is

isomorphic to the product of two sets. Contrarily to the union and intersection, the order of the family is important.

```
Definition productt (In:Set) (f:In->Set):=
  Zo (powerset (product In (uniont f)))
  (fun z => fgraph z & domain z = In & forall i:In, inc (V (Ro i) z) (f i)).

Definition productb g:= productt (fun x:domain g => V (Ro x) g).
Definition productf sf f:=  productb (L sf f).
```

We list below some basic properties of products.

```
Lemma productb_pr: forall f x,  fgraph f ->
  inc x (productb f) =
  (fgraph x &  domain x = domain f &
    forall i, inc i (domain x) -> inc (V i x) (V i f)).
Lemma productt_pr: forall (In:Set) (f:In->Set) x,
  inc x (productt f) =
  (fgraph x &  domain x = In  & forall i:In, inc (V (Ro i) x) (f i)).
Lemma productf_pr: forall sf f x,
  inc x (productf sf f) =
  (fgraph x &  domain x = sf &
    forall i, inc i (domain x) -> inc (V i x) (f i)).
```

We give here an extensionality properties for elements of a product.

```
Lemma productt_extensionality: forall (In:Set) (f:In->Set) x x',
  inc x (productt f) -> inc x' (productt f) ->
  (x = x') = (forall i:In,  V (Ro i) x = V (Ro i) x').
Lemma productb_extensionality: forall f x x',
  fgraph f ->
  inc x (productb f) -> inc x' (productb f) ->
  (x = x') = (forall i, inc i (domain f) -> V i x = V i x').
Lemma productf_extensionality: forall sf f x x',
  inc x (productf sf f) -> inc x' (productf sf f) ->
  (x = x') = (forall i, inc i sf -> V i x = V i x').
```

We define now pr_ι, the ι-th projection of a product; it is like pr_1 and pr_2 for the product of two sets. Let f be an element of the product and ι an index; we have $\mathrm{pr}_\iota f = f_\iota$, where f_ι denotes $f(\iota)$. Thus this mapping is nothing else than $\mathscr{V}$. Here we define a function, whose source is the product $\prod X_\kappa$ and whose target is X_ι.

Note. Since we have three types of products, we have three types of partial products, and three theorems for each property; For this reason, we just state basic properties of *pr_it*, and use only *pr_i*.

```
Definition pr_i f i:=  BL(V i) (productb f) (V i f).
Definition pr_it (In:Set) (f:In ->Set):=  pr_i (gbcreate f).

Lemma axioms_pri: forall f i,
  fgraph f -> inc i (domain f) ->
  transf_axioms (V i)(productb f)(V i f).
Lemma function_pri: forall f i,
  fgraph f -> inc i (domain f) ->
  is_function (pr_i f i).
```

```
Lemma W_pri: forall f i x,
  fgraph f -> inc i (domain f) -> inc x (productb f) ->
  W x (pr_i f i) = V i x.

Lemma axioms_prit: forall (In:Set) (f:In->Set) i,
  transf_axioms (V (Ro i))(producttt f)(f i).
Lemma function_prit: forall (In;Set) (f:In->Set) (i:In),
  is_function (pr_it f (Ro i)).
Lemma W_prit: forall (In:Set) (f:In ->Bet) (i:In) x,
  inc x (producttt f) -> W x (pr_it f (Ro i)) = V (Ro i) x.
```

If the sets X_ι are non empty, so is the product, and conversely. The idea is that we have $x_\iota \in X_\iota$ for some x_ι, and we can construct a function $\iota \mapsto x_\iota$.

```
Lemma trivial_product :
  productb emptyset = singleton emptyset.
Lemma trivial_function: forall f, L emptyset f = emptyset.
Lemma trivial_product2: forall f,
  productb (L emptyset f) = singleton emptyset.
Lemma nonempty_product: forall f,
  fgraph f ->  (forall i, inc i (domain f) -> nonempty (V i f)) ->
  nonempty (productb f).
Lemma nonempty_product2: forall f,
 fgraph f ->  nonempty (productb f) ->
  (forall i, inc i (domain f) -> nonempty (V i f)).
Lemma nonempty_producttt: forall (In:Set) (f:In->Set),
  (forall i, nonempty (f i)) ->  nonempty (producttt f).
Lemma nonempty_producttt2: forall (In:Set) (f:In->Set),
  nonempty (producttt f) -> (forall i, nonempty (f i)).
```

Assume $X_\iota \subset E$. An element of the product $\prod_{\iota\in I} X_\iota$ is the graph of a function from I to E. The converse is true if $X_\iota = E$ for all ι. Then $\prod_{\iota\in I} E = E^I$.

```
Lemma set_of_gfunctions_pr1: forall a b z,
  inc z (set_of_gfunctions a b) =
  (fgraph z & domain z = a & sub z (product a b)).
Lemma set_of_gfunctions_pr2: forall a b z,
  inc z (set_of_gfunctions a b) =
  (fgraph z & domain z = a & sub (range z) b).
Lemma sub_product_graphset:  forall f x,
  fgraph f -> sub (unionb f) x ->
  sub (productb f) (set_of_gfunctions (domain f) x).
Lemma eq_product_graphset:  forall f x,
  fgraph f -> (forall i, inc i(domain f) -> V i f = x) ->
  productb f = set_of_gfunctions (domain f) x.
```

¶ Special cases of products. We have already seen that if the index set I is empty, then the product has a single element: the empty graph. We consider here the case where the index set has one element α. The product is then isomorphic to X_α. The definition *product1a* constructs a product with one set given the parameters $x = X_\alpha$ and $a = \alpha$. If $I = \{\alpha\}$, then the product is equal to x^I since I is a singleton. We construct a function from x into the product that associates to each y the constant function y.

```
Definition product1 (x a:Set) := producttt  (fun _:singleton a => x).
Lemma product1_pr: forall x a,
```

```
  product1 x a= set_of_gfunctions (singleton a) x.
Lemma inc_product1: forall x a y,
  inc y (product1 x a) = (fgraph y & domain y = singleton a
    & inc (V a y) x).
Definition product1_canon x a :=
  BL (fun i : Set => L(singleton a) (fun _ : Set => i))  x (product1 x a).
Lemma transf_axioms_product1_canon: forall x a,
 transf_axioms (fun i =>  L(singleton a) (fun _ : Set => i)) x (product1 x a).
Lemma function_product1_canon: forall x a,
  is_function (product1_canon x a).
Lemma W_product1_canon: forall x a i,
  inc i x -> W i (product1_canon x a) =
  L(singleton a) (fun _ : Set => i).
Lemma bijective_product1_canon: forall x a,
  bijective(product1_canon x a).   (* 15 *)
```

We now consider the case of two sets. For each index set $I = \{\alpha, \beta\}$, if x and y are two sets, we can consider the $(X_\iota)_{\iota \in I}$ such that $x = X_\alpha$, $y = X_\beta$. We can define a bijection between $x \times y$ and the the product $\prod X_\iota$. For simplicity, consider only the case where I is the basic doubleton (we might as well consider the case where $\alpha = \emptyset$ and $\beta = \{\emptyset\}$, which is what Bourbaki suggests whenever two distinct sets are needed).

```
Definition product2 x y :=
  productf two_points (variant TPa x y).
Definition product2_canon x y :=
  BL (fun z => (Lvariantc (P z) (Q z)))  (product x y) (product2 x y).

Lemma inc_product2: forall x y z,
  inc z (product2 x y) = (fgraph z & domain z = two_points &
    inc (V TPa z) x & inc (V TPb z) y).
Lemma transf_axioms_product2_canon: forall x y,
  transf_axioms (fun z  => Lvariantc (P z) (Q z))
  (product x y) (product2 x y).
Lemma function_product2_canon: forall x y,
  is_function (product2_canon x y).
Lemma W_product2_canon: forall x y z,
  inc z (product x y) -> W z (product2_canon x y) = Lvariantc (P z) (Q z).
Lemma bijective_product2_canon: forall x y,
  bijective (product2_canon x y).  (* 16 *)
```

If each X_ι is a singleton, so is the product $\prod X_\iota$.

```
Definition is_singleton x := exists u, x = singleton u.
Lemma is_singleton_pr: forall x,
  is_singleton x = (nonempty x & (forall a b, inc a x -> inc b x -> a = b)).
Lemma singleton_product: forall f,
  fgraph f -> (forall i, inc i (domain f) -> is_singleton (V i f))
  -> is_singleton (productb f).
```

The set of graphs of constant functions $I \to E$ is called the diagonal of E^I. The application that associates to x the constant function with value x is an injection from E to E^I.

```
Definition constant_graph s x :=
  L s (fun _ => x).
```

```
Definition is_constant_graph f :=
  fgraph f &
  (forall x x', inc x (domain f) -> inc x' (domain f) -> V x f = V x' f).
Definition diagonal_graphp e i :=
  Zo(set_of_gfunctions i e) is_constant_graph.
Definition constant_functor i e:=
  BL(fun x => constant_graph i x) e (set_of_gfunctions i e).

Lemma constant_graph_function: forall f,
  is_constant_function f -> is_constant_graph(graph f).
Lemma V_constant_graph: forall s x y,
  inc y s -> V y (constant_graph s x) = x.
Lemma constant_graph_small_range: forall f,
  is_constant_graph f -> small_set(range f).
Lemma diagonal_graph_pr: forall e i x,
  inc x (diagonal_graphp e i) =
  (is_constant_graph x & domain x = i & sub (range x) e).
Lemma constant_graph_is_constant: forall x y,
  is_constant_graph(constant_graph x y).
Lemma injective_cf : forall i e,
  nonempty i -> injective (constant_functor i e).
```

Proposition 4 [2, p. 104] says: Given a family X_ι and a bijection f, the product $\prod X_\iota$ is isomorphic to the product $\prod X_{f(\iota)}$. Note that in the case of union or intersection, we have equality if f is surjective. The idea is that, if $x_\iota \in X_\iota$ and $\iota = f(\kappa)$ then $(x \circ f)_\kappa \in (X \circ f)_\kappa$. Some machinery is needed because x is a graph and f a function. These objects are not composable (we must compose x and the graph of f).

```
Definition product_compose f u :=
  BL (fun x => >compose_graph x (graph u))
  (productb f) (productf (source u) (fun k => V (W k u) f)).
Lemma compose_V: forall u v x,
  fgraph u -> fgraph v -> fgraph (compose_graph u v) ->
  inc x (domain (compose_graph u v)) ->
  V x (compose_graph u v) = V (V x v) u.

Lemma axioms_pc0: forall f u c,
  fgraph f -> bijective u -> target u = domain f ->
  inc c (productb f)  ->
  let g:= compose (corresp (domain c) (range c) c) u  in
    is_function g & compose_graph c (graph u) = graph g
    (forall i, inc i (source u) ->
      V i (graph g) = V (V i (graph u)) c).
Lemma axioms_pc2: forall f u c,
  fgraph f -> bijective u -> target u = domain f ->
  inc c (productb f)  -> fgraph (compose_graph c (graph u)).
Lemma axioms_pc: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  transf_axioms (fun x => compose_graph x (graph u))
     (productb f) (productf (source u) (fun k  => V (W k u) f)).
Lemma function_pc: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  is_function(product_compose f u).
Lemma W_pc: forall f u x,
  fgraph f -> bijective u -> target u = domain f ->
  inc x (productb f) -> W x (product_compose f u) = compose_graph x (graph u).
```

```
Lemma WV_pc: forall f u x i,
  fgraph f -> bijective u -> target u = domain f ->
  inc x (productb f) -> inc i (source u) ->
   V i (W x (product_compose f u)) = V  (W i u) x.
Lemma bijective_pc: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  bijective (product_compose f u).   (* 24 *)
```

6.4 Partial products

Given a family X_i with index I and a subset $J \subset I$, we can restrict the family to J; we have $\bigcup_J \subset \bigcup_I$ and $\bigcap_J \supset \bigcap_I$. The case of a product is more complicated. If $x \in \prod_I$, the restriction of x to J is in $\prod_J$. The converse is not clear: given an element of $\prod_J$, is there an extension? Is it unique? We start with some lemmas concerning restrictions.

```
Lemma graph_extensionality: forall r r',
  is_graph r -> is_graph r' ->
  (r = r') =  forall u v, related r u v = related r' u v.
Lemma restriction_graph1 : forall x f,
  fgraph f -> sub x (domain f) ->  L x (fun i  => V i f) = restr f x.
Lemma restriction_graph2 : forall f j,
  fgraph f -> sub j (domain f) ->
  L j (fun x  => V x f) = compose_graph f (diagonal j).
Lemma restr_domain1 : forall f x,
  fgraph f -> sub x (domain f) -> domain (restr f x) =  x.
```

We now define the restriction product and the function that associates to each x of the product its restriction to J. This function will be denoted by pr_J.

```
Definition restriction_product f j := productb (restr f j).
Definition pr_j f j :=
  BL (restriction_graph j) (productb f)(restriction_product f j).
Lemma axioms_prj: forall f j,
  fgraph f -> sub j (domain f) ->
  transf_axioms (restriction_graph j)
  (productb f)(restriction_product f j).
Lemma function_prj: forall f j,
  fgraph f -> sub j (domain f) -> is_function (pr_j f j).
Lemma W_prj: forall f j x,
  fgraph f -> sub j (domain f) -> inc x (productb f)
  -> W x (pr_j f j) = (restriction_graph j x).
Lemma WV_prj: forall f j x i,
  fgraph f -> sub j (domain f) -> inc x (productb f) -> inc i j
  -> V i (W x (pr_j f j))  =  V i x.
```

Propositions 6 and 5 [2, p. 105] state that if X_ι is nonempty for $\iota \notin J$, then we can extend a function defined on J to the whole of I (using the axiom of choice or the fact that a nonempty product is nonempty). Then pr_J is surjective. A special case is when J has a single element α. Then pr_J is the composition of pr_α and the canonical function that identifies a product of a single set with this set. Thus pr_α is surjective. A consequence is that if $X_\iota \subset Y_\iota$ then $\prod X_\iota \subset \prod Y_\iota$ (the converse is true if no X_ι is empty).

```
Theorem prolongation_exists: forall f j g,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  fgraph g -> domain g = j -> sub j (domain f) ->
  (forall i, inc i j -> inc (V i g) (V i f)) ->
  exists h, domain h = domain f &
    fgraph h & (forall i, inc i (domain f) -> inc (V i h) (V i f)) &
    (forall i, inc i j -> V i h  = V i g).    (* 35 *)
Theorem surjective_prj: forall f j,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  sub j (domain f) -> surjective (pr_j f j).
Lemma surjective_pri: forall f k,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  inc k (domain f) -> surjective (pr_i f k).     (* 34 *)

Lemma productb_monotone1: forall f g,
  fgraph f -> fgraph g  -> domain f = domain g ->
  (forall i, inc i (domain f) -> sub (V i f) (V i g))
  -> sub (productb f) (productb g).
Lemma productb_monotone2: forall f g,
  fgraph f -> fgraph g  -> domain f = domain g ->
  (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  sub (productb f) (productb g)  ->
  (forall i, inc i (domain f) -> sub (V i f) (V i g)).
```

6.5 Associativity of products of sets

Consider a family X_ι. Assume that the index set I is the union of sets J_λ. For each λ, we can consider the function pr_{J_λ}. If $f \in \prod X_\iota$, then $\mathrm{pr}_{J_\lambda} f \in \prod_{\iota\in J_\lambda}$. We can consider this as a function of λ and write it as $(\mathrm{pr}_{J_\lambda} f)_\lambda$. Thus we get a function

$$f \mapsto (\mathrm{pr}_{J_\lambda} f)_{\lambda\in L} \qquad \prod_{\iota\in I} X_\iota \to \prod_{\lambda\in L} \big(\prod_{\iota\in J_\lambda} X_\iota\big)$$

It is a bijection if the sets J_λ are mutually disjoint, in other words if they form a partition of I. This is Proposition 7 [2, p. 106].

```
Definition axioms_prod_assoc f g :=
  fgraph f  & partition_fam g (domain f).

Definition prod_assoc_map f g :=
  BL(fun z => (L (domain g) (fun l => W z (pr_j f (V l g)))))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma axioms_pam: forall f g,
  axioms_prod_assoc f g ->
  transf_axioms(fun z => (L (domain g) (fun l => W z (pr_j f (V l g)))))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma function_pam: forall f g,
  axioms_prod_assoc f g ->
  is_function (prod_assoc_map f g).
Lemma W_pam: forall f g x,
  axioms_prod_assoc f g -> inc x (productb f) ->
  W x (prod_assoc_map f g) =  (L (domain g) (fun l => W x (pr_j f (V l g)))).
Lemma injective_pam: forall f g,
```

```
  axioms_prod_assoc f g ->
  injective(prod_assoc_map f g).  (* 14 *)
Theorem bijective_pam: forall f g,
  axioms_prod_assoc f g ->
  bijective(prod_assoc_map f g).    (* 43 *)
```

Assume that the domain I is the disjoint union of two set I_1 and I_2. Let Y, Y_1 and Y_2 be the products of the family X_i over I, I_1 and I_2. There is a bijection between Y and $Y_1 \times Y_2$, because this set is equipotent to the product of the family with two elements. Assume now that each X_i is a singleton when $i \in I_2$. Then Y_2 is a singleton. The first projection from $Y_1 \times Y_2$ onto Y_1 is then a bijection. This gives a bijection between Y and Y_1. The last lemma here says that this bijection is pr_{I_1}.

```
Lemma Lvariantc_prop: forall x y,
  Lvariantc x y = L two_points (variant TPa x y).
Lemma prod_assoc_map2: forall f g,   (* 16 *)
  axioms_prod_assoc f g -> domain g = two_points
  -> equipotent (productb f)
  (product (restriction_product f (V TPa g)) (restriction_product f (V TPb g))).

Lemma bijective_p: forall x y,
  is_singleton y -> bijective (first_proj (product x y)).
Lemma bijective_prj_aux1: forall f j,
  fgraph f -> sub j (domain f) ->
  (forall i, inc i (complement (domain f) j) -> is_singleton (V i f)) ->
  is_singleton (restriction_product f (complement (domain f) j)).
Lemma bijective_prj_aux2: forall f g,
  two_points = domain g ->
  target (prod_assoc_map f g) =
  source (inverse_fun (product2_canon  (restriction_product f (V TPa g))
  (restriction_product f (V TPb g)))).
Lemma bijective_prj: forall f j,
  fgraph f -> sub j (domain f) ->
  (forall i, inc i (complement (domain f) j) -> is_singleton (V i f)) ->
  bijective (pr_j f j).      (* 54 *)
```

6.6 Distributivity formulae

Let $((X_{\lambda,\iota})_{\iota\in J_\lambda})_{\lambda\in L}$ be a family of families of sets. Let $I = \prod J_\lambda$. We assume that L and I are not empty. We have (Proposition 8 [2, p. 107])

$$\bigcup_{\lambda\in L}\Big(\bigcap_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcap_{f\in I}\Big(\bigcup_{\lambda\in L} X_{\lambda,f(\lambda)}\Big),$$

$$\bigcap_{\lambda\in L}\Big(\bigcup_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcup_{f\in I}\Big(\bigcap_{\lambda\in L} X_{\lambda,f(\lambda)}\Big).$$

The first result can be shown as follows. If x is in the LHS, there is a λ such x is in the intersection over J_λ, hence $x \in X_{\lambda,\mu}$, whatever μ; in particular it could be $f(\lambda)$. Conversely, Bourbaki assumes that x is not in the LHS; he considers the set $\{\iota \in J_\lambda \mid x \notin X_{\lambda,\iota}\}$. This set is not empty so that there is a function $f \in I$ whose value is in the set, so that x cannot be in the union of $X_{\lambda,f(\lambda)}$. The second result is shown by taking complements in a big set, namely the union of all sets involved.

```
Theorem distrib_union_inter: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  unionf (domain f) (fun l => intersectionb (V l f)) =
  intersectionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (unionf (domain f) (fun l => V (V l g) (V l f)))).       (* 24 *)
Lemma distrib_inter_union: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  intersectionf (domain f) (fun l => unionb (V l f)) =
  unionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (intersectionf (domain f) (fun l => V (V l g) (V l f)))).   (* 87 *)
```

A corollary is when L has two elements; in the original version, we used the property that a pair is a set with two distinct elements; here we use the canonical doubleton. Consider two families $F = (F_\iota)_{\iota\in I}$ and $G = (G_\kappa)_{\kappa\in K}$, and the function L that maps the canonical doubleton to $\{F,G\}$. Then $L(\iota)$ is a functional graph with non-empty domain, provided the same holds for F and G. The distributivity formulas use the set $I = \prod J_\lambda$. This set is the product of the domains of the graphs $L(\iota)$, namely I and J. We show that it is the product of the family with index set (F,G), that maps the first element to I and the second to J.

```
Lemma fgraph_rec_Lvariantc: forall f g,
  fgraph f -> fgraph g ->
  (forall l, inc l (domain (Lvariantc f g)) ->
    fgraph (V l (Lvariantc f g))).
Lemma nonempty_rec_dom_Lvariantc: forall f g,
  nonempty (domain f) -> nonempty (domain g) ->
  forall l, inc l (domain (Lvariantc f g)) ->
    nonempty (domain (V l (Lvariantc f g))).
Lemma product_Lvariantc: forall f g,
  productf (domain (Lvariantc f g)) (fun l => domain (V l (Lvariantc f g))) =
  product2 (domain f) (domain g).
```

The result is now the following: the union of $\bigcap_{\iota\in I} F_\iota$ and $\bigcap_{\kappa\in K} G_\kappa$ is the intersection on L of all $F_\iota \cup G_\kappa$; there is a similar formula if we exchange union and intersection. The general distributivity formula says that L is some complicated product, but it can be replaced by an equivalent set; we use the fact that the product of the family of two sets is equipotent to a normal product, so that $L = I \times K$.

```
Lemma distrib_union2_inter: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  union2 (intersectionb f)(intersectionb g) =
  intersectionf(product (domain f)(domain g)) (fun z =>
    (union2 (V (P z) f) (V (Q z) g))).     (* 55 *)
Lemma distrib_inter2_union: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  intersection2 (unionb f)(unionb g) =
  unionf(product (domain f)(domain g)) (fun z =>
    (intersection2 (V (P z) f) (V (Q z) g))).     (* 50 *)
```

We say here that the union and intersection of a singleton {X} is X. We also gives additional properties of the empty set. These will be used in the next theorem for exceptional situations.

```
Lemma unionf_singleton: forall x f, unionf (singleton x) f =  f x.
Lemma intersectionf_singleton: forall x f, intersectionf (singleton x) f =  f x.
Lemma empty_function: forall f, L emptyset f = emptyset.
Lemma trivial_product1: forall f, productf emptyset f = singleton emptyset.
Lemma unionf_emptyset: forall f, unionf emptyset f =  emptyset.
Lemma nonempty_product3: forall sf f i,
  inc i sf -> f i = emptyset ->  productf sf f = emptyset.
```

Let $((X_{\lambda,\iota})_{\iota\in J_\lambda})_{\lambda\in L}$ be a family of families of sets. Let $I = \prod J_\lambda$. We assume L and I not empty in the case of intersection. Proposition 9 [2, p. 109] says

$$\prod_{\lambda\in L}\Big(\bigcup_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcup_{f\in I}\Big(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\Big)$$

$$\prod_{\lambda\in L}\Big(\bigcap_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcap_{f\in I}\Big(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\Big)$$

In the case of union, we have to consider the special cases where L and I could be empty. Otherwise the proofs are similar. We must sometimes find an element f in the product I such that $f(\lambda)$ satisfies a given property $P(\lambda)$. We do this by considering the representative of the non-empty set $\{\lambda \in L, P(\lambda)\}$.

```
Theorem distrib_prod_union: forall f,
  fgraph f ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  productf (domain f) (fun l => unionb (V l f)) =
  unionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))).     (* 29 *)
Theorem distrib_prod_intersection: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  productf (domain f) (fun l => intersectionb (V l f)) =
  intersectionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))).    (* 26 *)
```

Let X_λ be the union of $X_{\lambda,\iota}$. The distributivity formula says that the product $\prod X_\lambda$ is a union; this union is a partition of the product, provided that the sets are mutually disjoint, i.e., if the $X_{\lambda,\iota}$ form a partition of X_λ.

```
Lemma partition_product: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> (partition_fam (V l f) (unionb (V l f)))) ->
  partition_fam(L(productf (domain f) (fun l => domain (V l f)) )
    (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))))
  ( productf (domain f) (fun l => unionb (V l f))).    (* 19 *)
```

We apply the distributivity formulas to the case of two families of sets. In a first variant, we consider the product of a family of two sets, after that, we convert it to a normal product.

```
Lemma distrib_prod2_union: forall f g,
  fgraph f -> fgraph g ->
  nonempty (domain f) -> nonempty (domain g) ->
  product2 (unionb f)(unionb g) =
```

```
  unionf(product (domain f)(domain g))
      (fun z =>  (product2 (V (P z) f) (V (Q z) g))). (* 55 *)
Lemma distrib_prod2_inter: forall f g,
  fgraph f -> fgraph g ->
  nonempty (domain f) -> nonempty (domain g) ->
  product2 (intersectionb f)(intersectionb g)=
  intersectionf(product (domain f)(domain g)) (fun z =>
    (product2 (V (P z) f) (V (Q z) g))).   (* 54 *)

Lemma distrib_product2_union: forall f g,
  fgraph f -> fgraph g ->
  product (unionb f)(unionb g) =
  unionf(product (domain f)(domain g)) (fun z =>
    (product (V (P z) f) (V (Q z) g))).               (* 34 *)
Lemma distrib_product2_inter: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  product (intersectionb f)(intersectionb g) =
  intersectionf(product (domain f)(domain g)) (fun z =>
    (product (V (P z) f) (V (Q z) g))).              (* 34 *)
```

Proposition 10 [2, p. 110] says that the intersection of a product is the product of the intersection.

$$\prod_{\iota\in I}\Big(\bigcap_{\kappa\in K} X_{\iota,\kappa}\Big) = \bigcap_{\kappa\in K}\Big(\prod_{\iota\in I} X_{\iota,\kappa}\Big)$$

This is a special case of the general distributivity formula where the set J_λ is independent of λ. In the case of two families of two sets, we get $(a\times b)\cap(c\times d) = (a\times c)\cap(b\times d)$. In the case of union, the general theorem says $(a\times b)\cup(c\times d) = (a\times c)\cup(b\times d)\cup(a\times d)\cup(b\times c)$ and there is no simpler formula.

```
Theorem distrib_inter_prod: forall f sa sb,
  fgraph f -> nonempty sb ->
  intersectionf sb (fun k => productf sa (fun i=> V (J i k) f)) =
  productf sa (fun i => intersectionf sb (fun k=> V (J i k) f)).     (* 25 *)
```

If one of the sets I or K is a doubleton then we get

$$\Big(\prod_{\iota\in I} X_\iota\Big)\cap\Big(\prod_{\iota\in I} Y_\iota\Big) = \prod_{\iota\in I}(X_\iota\cap Y_\iota),\qquad \Big(\bigcap_{\iota\in I} X_\iota\Big)\times\Big(\bigcap_{\iota\in I} Y_\iota\Big) = \bigcap_{\iota\in I}(X_\iota\times Y_\iota).$$

```
Lemma productf_extension : forall sf1 f1 sf2 f2,
  L sf1 f1 = L sf2 f2 -> productf sf1 f1 = productf sf2 f2.

Lemma distrib_prod_inter2_prod: forall f g,
  fgraph f -> fgraph g -> domain f = domain g ->
  nonempty (domain f) ->
  intersection2 (productb f) (productb g) =
  productf (domain f) (fun i => intersection2 (V i f) (V i g)).  (* 25 *)

Lemma distrib_inter_prod_inter: forall f g,
    fgraph f -> fgraph g -> domain f = domain g ->
    nonempty (domain f) ->
    product2 (intersectionb f) (intersectionb g) =
    intersectionf (domain f) (fun i => product2 (V i f) (V i g)). (* 23 *)
```

Given two functional graphs f and f' with the same domain I, we define the product to be the graph that associates $(f(x), f'(x))$ to x. Let $f'' = (f, f')$ be a pair of graphs; we can

consider it as a function that associates $(f(x), f'(x))$ to x. Thus we have a mapping from $\prod \mathrm{F}_\iota \times \prod \mathrm{F}'_\iota$ into $\prod(\mathrm{F}_\iota \times \mathrm{F}'_\iota)$. We need a bunch of lemmas in order to prove that this mapping is a bijection.

```
Definition prod_of_function x x':=
  L(domain x)(fun i => J (V i x) (V i x')).

Definition prod_of_products_canon f f':=
  BL(fun w => prod_of_function (P w) (Q w))
  (product (productb f) (productb f'))
  (productf (domain f)(fun i => product (V i f) (V i f'))).

Definition prod_of_product_aux f f' :=
  fun i =>  (product (W i f) (W i f')).

Definition prod_of_prod_target f f' :=
  fun_image(source f)(prod_of_product_aux f f').

Definition prod_of_products f f' :=
  BL (prod_of_product_aux f f')(source f)(prod_of_prod_target f f').

Lemma inc_prod_of_prod_target: forall f f' x,
  inc x (prod_of_prod_target f f') =
  (exists i, inc i (source f) & product (W i f) (W i f') = x).
Lemma function_prod_of_products: forall f f',
  is_function (prod_of_products f f').
Lemma source_prod_of_products: forall f f',
  source (prod_of_products f f') = source f.
Lemma target_prod_of_products: forall f f',
  target (prod_of_products f f') =  prod_of_prod_target f f'.
Lemma W_prod_of_products: forall f f' i,
  inc i (source f) ->
  W i (prod_of_products f f') =  product (W i f) (W i f').
Lemma prod_of_products_fam_pr: forall f f' x,
  is_function x -> source x = source f ->
  target x = union (prod_of_prod_target f f') ->
  inc (graph x) (productb (graph (prod_of_products f f'))) =
  forall i, inc i (source f) ->
    (is_pair (W i x) &  inc (P (W i x)) (W i f) &  inc (Q (W i x)) (W i f')).
Lemma axioms_prod_of_function:forall x x' f f',
  is_function f -> is_function f' -> source f = source f' ->
  inc (graph x) (productb (graph f)) ->  inc (graph x') (productb (graph f')) ->
  transf_axioms (fun i => J (W i x) (W i x'))
     (source f) (union (prod_of_prod_target f f')).
Lemma W_prod_of_function:forall x x' f f' i,
  is_function f -> is_function f' -> source f = source f' ->
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc i (source f) ->
  V i (prod_of_function x x') =  J (V i x) (V i x').
Lemma function_prod_of_function: forall x x' f f',
  is_function f -> is_function f' -> source f = source f' ->
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc (prod_of_function x x')
  (productb (graph (prod_of_products f f'))).   (* 17 *)
Lemma target_popc_aux:forall f f',
 is_function f -> is_function f' -> source f = source f' ->
```

```
 productb(L (domain (graph f))
   (fun i => product (V i (graph f)) (V i (graph f')))) =
 productb(graph (prod_of_products f f')).
Lemma target_popc:forall f f',
  is_function f -> is_function f' -> source f = source f' ->
  target (prod_of_products_canon (graph f) (graph f')) =
  (productb (graph (prod_of_products f f'))).
Lemma axioms_popc:forall f f',
 is_function f -> is_function f' -> source f = source f' ->
  transf_axioms(fun w => prod_of_function (P w) (Q w))
  (product (productb (graph f)) (productb (graph f')))
  (productb (graph (prod_of_products f f'))).
Lemma W_popc:forall f f' w,
  is_function f -> is_function f' -> source f = source f' ->
  inc w (product (productb (graph f)) (productb (graph f'))) ->
  W w (prod_of_products_canon (graph f) (graph f')) =
  prod_of_function (P w) (Q w).

Lemma bijection_popc: forall f f',
  is_function f -> is_function f' -> source f = source f' ->
  bijective (prod_of_products_canon (graph f) (graph f')).    (* 74 *)
```

6.7 Extensions of mappings to products

$$\begin{array}{ccc} X_\iota & \xrightarrow{f_\iota} & Y_\iota \\ {\scriptstyle \mathrm{pr}_\iota}\uparrow & & \uparrow{\scriptstyle \mathrm{pr}_\iota} \\ \prod X_\iota & \xrightarrow[(f_\iota)_{\iota\in I}]{} & \prod Y_\iota \end{array} \qquad \text{(extension)}$$

Assume that X_ι, Y_ι and f_ι are families with the same index I. We assume that f_ι is a functional graph with source X_ι and target Y_ι. If $x \in \prod X_\iota$ then $x_\iota \in X_\iota$, $f_\iota(x_\iota) \in Y_\iota$ and the mapping $\iota \mapsto f_\iota(x_\iota)$ is in $\prod Y_\iota$. This induces a function $\prod X_\iota \to \prod Y_\iota$ called the *extension* of the functions f_i.

```
Definition ext_map_prod_aux x  f := fun i=> V (V i x)  (f i).
Definition ext_map_prod In src trg f :=
  BL (fun x => L In (ext_map_prod_aux x f))
    (productf In src ) (productf In trg ).

Definition ext_map_prod_axioms In src trg f :=
  forall i, inc i In -> (fgraph (f i) & domain (f i) = src i &
    sub (range (f i)) (trg i)).

Lemma taxioms_ext_map_prod: forall In src trg f,
  ext_map_prod_axioms In src trg f ->
  transf_axioms (fun x => L In (ext_map_prod_aux x f))
    (productf In src) (productf In trg).
Lemma function_ext_map_prod: forall In src trg f,
  ext_map_prod_axioms In src trg f -> is_function ( ext_map_prod In src trg f).
Lemma W_ext_map_prod: forall In src trg f x,
  ext_map_prod_axioms In src trg f ->
  inc x (productf In src) ->
```

```
  W x (ext_map_prod In src trg f) = L In (ext_map_prod_aux x f).
Lemma WV_ext_map_prod: forall In src trg f x i,
  ext_map_prod_axioms In src trg f ->
  inc x (productf In src) -> inc i In ->
  V i (W x (ext_map_prod In src trg f)) = V (V i x)  (f i).
```

Proposition 11 [2, p. 111] says that composition of extensions is extension of compositions. Bourbaki uses this property to show that if all f_ι are injective, so is the extension, by exhibiting a left inverse. We use a direct proof because it is easier (note that f_ι is not a function, just the graph of a function).

```
Lemma composable_ext_map_prod: forall  In p1 p2 p3  g f h,
  ext_map_prod_axioms In p1 p2 f ->
  ext_map_prod_axioms In p2 p3 g ->
  (forall i, inc i In -> h i = fcompose (g i) (f i)) ->
  (forall i, inc i In -> fcomposable (g i) (f i)) ->
  ext_map_prod_axioms In p1 p3 h.

Lemma compose_ext_map_prod: forall  In p1 p2 p3  g f h,
  ext_map_prod_axioms In p1 p2 f ->
  ext_map_prod_axioms In p2 p3 g ->
  (forall i, inc i In -> h i = fcompose (g i) (f i)) ->
  (forall i, inc i In -> fcomposable (g i) (f i)) ->
  compose (ext_map_prod In p2 p3 g)  (ext_map_prod In p1 p2 f) =
  (ext_map_prod In p1 p3 h).      (* 27 *)

Lemma injective_ext_map_prod: forall  In p1 p2 f,
  ext_map_prod_axioms In p1 p2 f ->
  (forall i, inc i In -> injective_graph (f i)) ->
  injective (ext_map_prod In p1 p2 f).
Lemma surjective_ext_map_prod: forall  In p1 p2 f,
  ext_map_prod_axioms In p1 p2 f ->
  (forall i, inc i In -> range (f i) = p2 i) ->
  surjective (ext_map_prod In p1 p2 f).     (* 21 *)
```

Let f be a function from E to A, where A is a product X_ι over I. Consider the function $\mathrm{pr}_\iota \circ f$ from E to X_ι. Its extension to products is some function $\overline{f}$ from E^I to $\prod X_\iota$. Let d be the diagonal mapping from E to E^I. We have $f = \overline{f} \circ d$. If f_ι is a family of functions from E to X_ι, and $\overline{f}$ is its extension to the products, then $\mathrm{pr}_\iota \circ (\overline{f} \circ d) = f_\iota$. The mapping from f to $\overline{f}$ is a bijection between $(\prod X_\iota)^E$ and $\prod X_\iota^E$.

We prove the following facts one after the other. If f is a function from E to A, then $\mathrm{pr}_\iota \circ f$ is a function from E to X_ι. To $g \in A^E$ we associate a function from E to A. The mapping $\iota \mapsto G(\mathrm{pr}_\iota \circ f)$, where G denotes the graph of a function, is an element of $\prod X_\iota^E$. We have a function $A^E \to \prod X_\iota^E$. We define $\overline{f}$. We show $f = \overline{f} \circ d$. We show $\mathrm{pr}_\iota \circ (\overline{f} \circ d) = f_\iota$. We finally show that the mapping is bijective.

```
Definition fun_set_to_prod src F :=
  BL(fun f =>
       L(domain F)( fun i=> (graph (compose (pr_i F i)
          (corresp src (productb F) f)))))
     (set_of_gfunctions src (productb F))
    (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F)))).

Lemma fun_set_to_prod1: forall F f i,
```

```
  fgraph F -> inc i (domain F) ->
  is_function f -> target f = productb F ->
  (is_function (compose (pr_i F i) f) &
   source (compose (pr_i F i) f) = source f &
   target (compose (pr_i F i) f) = V i F&
   (forall x, inc x (source f) ->  W x  (compose (pr_i F i) f) = V i (W x f))).

Lemma fun_set_to_prod2: forall src  F f gf,
  fgraph F -> inc gf (set_of_gfunctions src (productb F)) ->
  f = (corresp src (productb F) gf) ->
  (is_function f & target f = productb F & source f = src).
Lemma fun_set_to_prod3: forall src  F,
  fgraph F -> transf_axioms(fun f =>
       L(domain F)( fun i=> (graph (compose (pr_i F i)
           (corresp src (productb F) f)))))
     (set_of_gfunctions src (productb F))
    (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F)))).
Lemma fun_set_to_prod4: forall src  F,
  fgraph F -> ( is_function (fun_set_to_prod src F)
   & source (fun_set_to_prod src F) = (set_of_gfunctions src (productb F))
   & target (fun_set_to_prod src F)
    = (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F))))).
Definition fun_set_to_prod5 F f :=
  ext_map_prod (domain F) (fun i=> source f)(fun i=> V i F)
  (fun i => (graph (compose (pr_i F i) f))).
Lemma fun_set_to_prod6: forall F f,    (* 40 *)
  fgraph F -> is_function f -> target f = productb F ->
  (is_function (fun_set_to_prod5 F f) &
    composable (fun_set_to_prod5 F f) (constant_functor (domain F)(source f) )&
    compose (fun_set_to_prod5 F f) (constant_functor (domain F)(source f)) =f).
Lemma fun_set_to_prod7: forall src F f g,
  fgraph F ->
  (forall i, inc i (domain F) -> inc (f i) (set_of_gfunctions src (V i F))) ->
  g = ext_map_prod (domain F) (fun i=> src)(fun i=> V i F) f ->
  (forall i, inc i (domain F) ->
    f i = graph (compose (pr_i F i) (compose g (constant_functor
           (domain F) src) ))).      (* 47 *)
Lemma fun_set_to_prod8: forall src  F,
  fgraph F -> bijective (fun_set_to_prod src F).      (* 58 *)
```

Chapter 7

Equivalence relations

The code of the first two sections of this chapter was originally written by Carlos Simpson[1]. He used *is_relation* where we now write *is_graph*. A relation between two objects x and y, often denoted by $x \sim y$, is a function of type $\mathscr{E} \to \mathscr{E} \to$ Prop. An *equivalence relation* will be a relation with some properties; an *equivalence* will be a graph with similar properties; this differs from the Bourbaki's definition, where an equivalence is a correspondence.

7.1 Definition of an equivalence relation

We say that x is related to y by the graph r and denote it by $x \overset{r}{\sim} y$ whenever the pair (x, y) is in the graph. The set of related objects is called the *substrate* of the relation.

```
Definition substrate r :=  union2 (domain r) (range r ).
```

We have some characterizations of the substrate. Only the last one requires that r be a graph.

```
Lemma inc_pr1_substrate : forall r y,  inc y r -> inc (P y) (substrate r).
Lemma inc_pr2_substrate : forall r y,  inc y r -> inc (Q y) (substrate r).
Lemma inc_arg1_substrate: forall r x y,  related r x y -> inc x (substrate r).
Lemma inc_arg2_substrate: forall r x y,  related r x y -> inc y (substrate r).
Lemma substrate_smallest : forall r s,
  (forall y, inc y r -> inc (P y) s) ->
  (forall y, inc y r -> inc (Q y) s) ->
  sub (substrate r) s.
Lemma inc_substrate_rw : forall r x,
  is_graph r -> inc x (substrate r) =
  ((exists y, inc (J x y) r) \/  (exists y, inc (J y x) r)).
```

We say that a property $\sim$ is *symmetric* if $a \sim b$ implies $b \sim a$, *transitive* if $a \sim b$ and $b \sim c$ implies $a \sim c$, *reflexive* on x if $a \in x$ is equivalent to $a \sim a$. We say that $\sim$ is an *equivalence relation* if it is symmetric and transitive. We say that it is an *equivalence relation on* E if moreover it is reflexive on E.

```
Definition reflexive_r (r:EEP) x :=
  forall y, inc y x = r y y.
```

[1] http://math.unice.fr/~carlos/themes/verif.html

```
Definition symmetric_r (r:EEP) :=
  forall x y, r x y -> r y x.
Definition transitive_r (r:EEP) :=
  forall x y z, r x y -> r y z -> r x z .
Definition equivalence_r (r:EEP) :=
  symmetric_r r & transitive_r r.
Definition equivalence_re (r:EEP) x :=
  equivalence_r r & reflexive_r r x.
```

The definitions for a graph are similar. We say that a graph is reflexive if its associated relation is reflexive on the substrate, i.e., if $x \overset{r}{\sim} y$ implies $x \overset{r}{\sim} x$ and $y \overset{r}{\sim} y$. An *equivalence* is a set that is reflexive, symmetric, and transitive.[2]

```
Definition is_reflexive r :=
  is_graph r & (forall y, inc y (substrate r) ->related r y y).
Definition is_symmetric r :=
  is_graph r  &  (forall x y, related r x y -> related r y x).
Definition is_transitive r :=
  is_graph r &
  (forall x y z, related r x y -> related r y z -> related r x z).
Definition is_equivalence r :=
  is_reflexive r & is_transitive r &  is_symmetric r.
```

Some trivial consequences of the definitions.

```
Lemma equivalence_is_graph : forall r, is_equivalence r -> is_graph r.
Lemma reflexive_inc_substrate : forall r x,
  is_reflexive r -> inc x (substrate r) = inc (J x x) r.
Lemma reflexive_ap :  forall r x,
  is_reflexive r -> inc x (substrate r) -> related r x x.
Lemma reflexive_ap2 :  forall r x y,
  is_reflexive r ->  related r x y -> related r x x.
Lemma symmetric_ap : forall r x y,
  is_symmetric r -> related r x y -> related r y x.
Lemma transitive_ap : forall r x z,
  is_transitive r ->
  (exists y, related r x y & related r y z) ->
  related r x z.
```

Some lemmas that show that if a graph is symmetric and transitive then it is reflexive.

```
Lemma symmetric_transitive_reflexive : forall r,
  is_symmetric r -> is_transitive r -> is_reflexive r.
Lemma show_Sequivalence_relation : forall r,
  is_graph r ->
  (forall x y, related r x y -> related r y x) ->
  (forall x y z, related r x y -> related r y z -> related r x z) ->
  is_equivalence r.
Lemma symmetric_transitive_equivalence : forall r,
  is_symmetric r -> is_transitive r -> is_equivalence r.
```

Some trivial properties of an equivalence.[3]

[2] In a previous version, we added the property *is_graph*, that can be deduced from the other ones

[3] Names changed in V3, was reflexivity or transitivity

```
Lemma reflexivity_e : forall r u,
  is_equivalence r -> inc u (substrate r) -> related r u u.
Lemma symmetricity : forall r u v,
  is_equivalence r -> related r u v -> related r v u.
Lemma transitivity_e : forall r u v w,
  is_equivalence r ->  related r u v -> related r v w -> related r u w.
```

In the case of an equivalence, the characterization of the substrate is easy.

```
Lemma domain_is_substrate: forall g,
 is_equivalence g -> domain g = substrate g.
Lemma substrate_sub : forall r s,
  sub r s -> sub (substrate r) (substrate s).
Lemma inc_substrate : forall r x,
  is_equivalence r ->
  inc x (substrate r) = (exists y, related r x y).
```

If r has some properties then $\overset{r}{\sim}$ has the corresponding ones.

```
Lemma reflexive_reflexive: forall r,
  is_reflexive r -> reflexive_r (related r) (substrate r).
Lemma symmetric_symmetric: forall r,
  is_symmetric r -> symmetric_r (related r).
Lemma transitive_transitive: forall r,
  is_transitive r -> transitive_r (related r).
Lemma equivalence_equivalence: forall r,
  is_equivalence r -> equivalence_re (related r)(substrate r).
```

We say that r is the graph of $\sim$ if $x \overset{r}{\sim} y$ is the same as $x \sim y$, whatever x and y. For every set E, we can define a set $r = g_{\mathrm{E}}(\sim)$, the graph of $\sim$ on E. This is the graph of some relation whose substrate is a subset of E.

```
Definition is_graph_of g (r:EEP):=
  forall u v, r u v = related g u v.

Definition graph_on (r:EEP) x := Zo(product x x)(fun w => r (P w)(Q w)).

Lemma is_graph_graph_on: forall r x,
  is_graph(graph_on r x).
Lemma substrate_graph_on: forall r x,
  sub (substrate (graph_on r x)) x.
```

Assume that $\sim$ is an equivalence relation on E. Then $g_{\mathrm{E}}(\sim)$ is the graph of $\sim$. The relation associated to this graph is $\sim$. Assume that $\sim$ is an equivalence relation that has a graph g, then it is an equivalence relation on the domain of g (which is also the substrate of g). This graph is an equivalence. We shall often use the fact that an equivalence relation on E is associated to an equivalence.

```
Lemma equivalence_has_graph0: forall r x,
  equivalence_re r x -> is_graph_of (graph_on r x) r.
Lemma related_graph_on: forall r x u v,
  equivalence_re r x ->
  r u v = related (graph_on r x) u v.
Lemma related_graph_on0: forall r x a b,
```

```
  inc (J a b) (graph_on r x)  = (inc a x & inc b x & r a b).
Lemma related_graph_on1: forall r x a b,
  related  (graph_on r x) a b = (inc a x & inc b x & r a b).
Lemma equivalence_has_graph: forall r x,
  equivalence_re r x -> exists g, is_graph_of g r.
Lemma equivalence_if_has_graph: forall r g,
  is_graph g ->  is_graph_of g r ->
  equivalence_r r -> equivalence_re r (domain g).
Lemma equivalence_if_has_graph2: forall r g,
  is_graph g ->   is_graph_of g r ->
  equivalence_r r -> is_equivalence g.
Lemma equivalence_has_graph2:forall r x,
  equivalence_re r x -> exists g,
    is_equivalence g & (forall u v, r u v = related g u v).
```

For Bourbaki, an *equivalence on a set* E is a correspondence whose source and target are both equal to E, and whose graph F is such that the relation $(x, y) \in F$ is an equivalence relation on E. Note: the correspondence is uniquely defined by F, since E is the substrate of F; conversely, given an equivalence F on E, the domain and range of F is E, thus $F \subset E \times E$, and (E, E, F) is a correspondence.[4]

```
(*
  Definition equivalence_cor r:=
    source r = target r &
    is_equivalence (graph r) & source r = (substrate (graph r)).
  Definition graph_to_eq_cor g := corresp (domain g)(domain g) g.
*)
```

If we take a set E and equality on E (i.e., the relation "$x \in E$ and $y \in E$ and $x = y$" between x and y), this gives an equivalence relation. The associated equivalence is the diagonal, and the associated correspondence is the identity function on E (note that this is the only case when the associated equivalence is a function). This is the equivalence with the smallest classes (we shall see that to each equivalence can be associated a partition, and partitions can be compared, so that equivalences can be compared. We have found the finest one).

```
Definition restricted_eq x := fun u => fun v => inc u x & u = v.

Lemma equality_equivalence: forall x,
  equivalence_re(restricted_eq x) x.
Lemma graph_of_restricted_eq: forall x,
  is_graph_of(diagonal x)(restricted_eq x).
Lemma equivalence_relation_diagonal: forall x,
  is_equivalence (diagonal x).
Lemma substrate_diagonal: forall x, substrate(diagonal x) = x.
```

We can consider the relation on E for which all elements are related. Its graph is $E \times E$.

```
Definition coarse u := product u u.

Lemma substrate_coarse : forall u, substrate (coarse u) = u.
Lemma related_coarse : forall u x y,
  related (coarse u) x y = (inc x u & inc y u).
Lemma equivalence_relation_coarse : forall u,
```

[4] This notion has been withdrawn in Version 3

```
  is_equivalence (coarse u).

Lemma is_graph_coarse: forall x, is_graph (coarse x).
Lemma inter2_is_graph1: forall x y, is_graph x ->
  is_graph (intersection2 x y).
Lemma inter2_is_graph2: forall x y, is_graph y ->
  is_graph (intersection2 x y).
raph: forall x y, is_graph x -> is_graph y ->
  is_graph (union2 x y).
Lemma sub_graph_coarse_substrate: forall r,
  is_graph r -> sub r (coarse (substrate r)).
```

Equipotency is an equivalence relation without a graph.

```
Lemma equivalence_equipotent: equivalence_r equipotent.
```

The fifth example in Bourbaki is: Suppose A $\subset$ E; then ($x \in$ E\A and $y = x$) or ($x \in$ A and $y \in$ A) is a equivalence on E.

```
Definition Bourbaki_ex5 a e:=
  Zo(product e e)(fun y=>
    ((inc (P y)(complement e a) & (P y = Q y)) \/ (inc (P y) a & inc(Q y) a))).

Lemma Bourbaki5_reflexive: forall a e x,
  inc x e -> related (Bourbaki_ex5 a e) x x.
Lemma substrate_ex5: forall a e,
  substrate (Bourbaki_ex5 a e) = e.
Lemma related_bourbaki_ex5: forall a e x y,
   inc x e -> inc y e -> (related (Bourbaki_ex5 a e) x y) =
   ((inc x(complement e a) & x = y) \/ (inc x  a & inc y a)).
Lemma equivalence_relation_bourbaki_ex5: forall a e,
  sub a e -> is_equivalence  (Bourbaki_ex5 a e).
```

Consider a non-empty family of relations $(\sim_i)_{i\in \mathrm{I}}$. We can consider the relation $\forall i, x \sim_i y$. This is equivalence (resp. an equivalence on E) if each $\sim_i$ is an equivalence (resp. an equivalence on E).

```
Lemma relation_intersection : forall z,
  nonempty z -> (forall r, inc r z -> is_graph r) ->
  is_graph (intersection z).
Lemma related_intersection : forall z x y,
  nonempty z ->
  related (intersection z) x y =
  (forall r, inc r z -> related r x y).
Lemma substrate_intersection_sub : forall r z,
  inc r z -> sub (substrate (intersection z)) (substrate r).
Lemma reflexive_intersection : forall z,
  nonempty z -> (forall r, inc r z -> is_reflexive r) ->
  is_reflexive (intersection z).
Lemma transitive_intersection : forall z,
  nonempty z -> (forall r, inc r z -> is_transitive r) ->
  is_transitive (intersection z).
Lemma symmetric_intersection : forall z,
  nonempty z -> (forall r, inc r z -> is_symmetric r) ->
  is_symmetric (intersection z).
```

```
Lemma equivalence_relation_intersection : forall z,
  nonempty z -> (forall r, inc r z -> is_equivalence r) ->
  is_equivalence (intersection z).
Lemma substrate_intersection : forall r e,
  nonempty r -> (forall z, inc  z r-> is_equivalence z) ->
  (forall z, inc z r -> substrate z = e) ->
  substrate (intersection r) = e.
```

We can consider the set of all equivalences on E. It is not empty.

```
Definition all_relations x :=  powerset (product x x).
Definition all_equivalence_relations x :=
  Zo (all_relations x) (fun r => (is_equivalence r)
    & (substrate r = x)).

Lemma inc_all_relations : forall r x,
  inc r (all_relations x) = (is_graph r & sub (substrate r) x).
Lemma inc_all_equivalence_relations : forall r x,
  inc r (all_equivalence_relations x) =
  (is_equivalence r & (substrate r = x)).
Lemma inc_coarse_all_equivalence_relations : forall u,
  inc (coarse u) (all_equivalence_relations u).
```

Proposition 1 [2, p. 114] says that a correspondence Γ between X and X is an equivalence on X if and only if X is the domain of Γ, $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. We prove this property for graphs rather than correspondences.

```
Lemma selfinverse_graph_symmetric: forall r,
  (is_symmetric r = (r= inverse_graph r)).
Lemma idempotent_graph_transitive: forall r,
  is_graph r-> (is_transitive r = sub  (compose_graph r r) r).
Theorem equivalence_pr: forall r,
  (is_equivalence r  = ((compose_graph r r) = r &  r= inverse_graph r)).
```

7.2 Equivalence classes; quotient set

Let f be a function on E; the relation $f(x) = f(y)$ is an equivalence relation on E. It has a graph $\mathrm{F}^{-1} \circ \mathrm{F}$, where F is the graph of f. We shall denote it by $\sim_f$.

```
Definition eq_rel_associated f :=
  fun x => fun y => (inc x (source f) & (inc y (source f)) & (W x f = W y f)).
Definition equivalence_associated f :=
  compose_graph (inverse_graph (graph f)) (graph f).

Lemma equivalence_ea: forall f,
  is_function f -> equivalence_re(eq_rel_associated f)(source f).
Lemma graph_of_ea: forall f,
  is_function f ->
  is_graph_of (equivalence_associated f) (eq_rel_associated f).
Lemma equivalence_graph_ea: forall f,
  is_function f->
  is_equivalence (equivalence_associated f).
Lemma substrate_graph_ea: forall f,
  is_function f->
```

```
  substrate (equivalence_associated f) = source f.
Lemma related_ea:forall f x y,
  is_function f ->
  related (equivalence_associated f) x y =
  (inc x (source f) & inc y (source f) &  W x f = W y f).
```

Bourbaki says that for every equivalence relation $\sim$ on E, there is a function f such that the equivalence associated with f is $\sim$ such that $\sim = \sim_f$. Let G be the graph of the equivalence relation and $x \in \mathrm{E}$. For $x \in \mathrm{E}$, the set $\mathrm{G}(x)$ of all y such that $(x, y) \in \mathrm{G}$ (also known as the cut of G at x) will be called the *equivalence class* of x, and the set of all equivalence classes will be called the *quotient set*, and denoted by $\mathrm{E}/\sim$ (or E/R if the relation is R). Let's denote by $\bar{x}$ the class of x modulo R, this is also $\mathrm{pr}_2\mathrm{G}_x$, where G_x denotes the set all elements $z \in \mathrm{G}$ with $\mathrm{pr}_1 z = x$. We shall use the four characteristic properties of classes: (a) $y \in \bar{x}$ if and only if $x \overset{\mathrm{R}}{\sim} y$, (b) $\bar{x} \subset \mathrm{E}$, (c) $x \in \mathrm{E}$ if and only if $\bar{x} \neq \emptyset$ and (d) if $x \in \mathrm{E}$, then $x \overset{\mathrm{R}}{\sim} y$ and $\bar{x} = \bar{y}$ are equivalent. This last property says that $\sim$ is the equivalence associated to the function $x \mapsto \bar{x}$.

```
Definition class (r x:Set) := fun_image (Zo r (fun z => P z = x)) Q.

Lemma inc_class : forall r x y,
  is_equivalence r -> inc y (class r x) = related r x y.
Lemma inc_class0 : forall r x y,
  is_equivalence r -> inc y (class r x) = related r x y.
Lemma class_is_cut: forall r x, is_equivalence r -> class r x = im_singleton r x.
Lemma sub_class_substrate: forall r x,
  is_equivalence r -> sub(class r x) (substrate r).
Lemma nonempty_class_symmetric : forall r x,
  is_equivalence r ->
  nonempty (class r x) = inc x (substrate r).
Lemma related_class_eq1: forall r u v, is_equivalence r ->
  related r u v -> class r u = class r v.
Lemma related_class_eq : forall r u v,
  is_equivalence r ->
  related r u u ->
  related r u v = (class r u = class r v).
```

We denote by $\hat{x} = \tau_z(z \in x)$ some element of x (if x is non-empty of course), so that $x \mapsto \hat{x}$ is is a mapping from E/R into E (it is a retraction of the canonical projection, meaning $x = \bar{\hat{x}}$). We say that x is *a class for* r if r is an equivalence (with substrate s_r), $\hat{x} \in s_r$ and $x = \bar{\hat{x}}$. Clearly, if $x \in \mathrm{E}$, then $\bar{x}$ is a class. Thus $a \overset{\mathrm{R}}{\sim} b$ if and only if that there is a class x such that $a \in x$ and $b \in x$. If $x \in \mathrm{E}/\mathrm{R}$ and $y \in x$ then $\hat{x} \overset{\mathrm{R}}{\sim} y$.

```
Definition quotient r := fun_image (substrate r) (class r).
Definition is_class r x := is_equivalence r
  & inc (rep x) (substrate r)  & x = class r (rep x).

Lemma inc_rep_itself:forall r x,  is_equivalence r ->
  inc x (quotient r) -> inc (rep x) x.
Lemma non_empty_in_quotient: forall r x,
  is_equivalence r -> inc x (quotient r) -> nonempty x.
Lemma is_class_class : forall r x, is_equivalence r ->
  inc x (substrate r) -> is_class r (class r x).
Lemma inc_quotient : forall r x,  is_equivalence r ->
  inc x (quotient r) = is_class r x.
```

```
Lemma in_class_related : forall r y z,
  is_equivalence r ->
  related r y z = (exists x, is_class r x & inc y x & inc z x).
Lemma related_rep_in_class:forall r x y,
  is_equivalence r -> inc x (quotient r) -> inc y x
  -> related r (rep x) y.
```

A class x is a nonempty subset of E such that for all $y \in x$, properties $z \in x$ and $y \overset{R}{\sim} z$ are equivalent. Two classes are equal or disjoint. If $x \in$ E then $\bar{x} \in$ E/R. If $x \in y$ and $y \in$ E/R then $x \in$ E. As a consequence, the union of E/R is E. If $x \in$ E/R then $\hat{x} \in$ E, and $\bar{\hat{x}} = x$. If $x \in$ E then $x \in \bar{x}$ and $x \overset{R}{\sim} \hat{\bar{x}}$. If u and v are in E/R, then $\hat{u} \overset{R}{\sim} \hat{v}$ if and only if $u = v$. The relation $u \overset{R}{\sim} v$ is equivalent to $u \in$ E and $v \in$ E and $\bar{u} = \bar{v}$. If $x \in y$ and $y \in$ E/R then $y = \bar{x}$.

```
Lemma is_class_rw : forall r x, is_equivalence r ->
  is_class r x = (nonempty x & sub x (substrate r) &
    (forall y z, inc y x -> (inc z x = related r y z)) ).
Lemma class_dichot : forall r x y,
  is_class r x -> is_class r y -> (x = y \/ disjoint x y).
Lemma inc_class_quotient : forall r x,
  is_equivalence r -> inc x (substrate r) ->
  inc (class r x) (quotient r).
Lemma inc_in_quotient_substrate : forall r x y,
  is_equivalence r -> inc x y -> inc y (quotient r)
  -> inc x (substrate r).
Lemma union_quotient : forall r,
  is_equivalence r ->   union (quotient r) = substrate r.
Lemma inc_rep_substrate : forall r x, is_equivalence r ->
  inc x (quotient r) -> inc (rep x) (substrate r).
Lemma class_rep : forall r x, is_equivalence r ->
  inc x (quotient r) -> class r (rep x) = x.
Lemma inc_itself_class : forall r x,
  is_equivalence r ->inc x (substrate r) ->  inc x (class r x).
Lemma related_rep_class : forall r x, is_equivalence r ->
  inc x (substrate r) -> related r x (rep (class r x)).
Lemma related_rep_rep : forall r u v,
  is_equivalence r -> inc u (quotient r) -> inc v (quotient r) ->
  related r (rep u) (rep v) = (u = v).
Lemma related_rw : forall r u v,
  is_equivalence r -> related r u v =
  (inc u (substrate r) & inc v (substrate r) & class r u = class r v).
Lemma is_class_pr: forall r x y,
  is_equivalence r -> inc x y -> inc y (quotient r)
  -> y = class r x.
```

The *canonical projection* is the mapping $x \mapsto \bar{x}$ from E onto E/R. An important property is that this function is surjective[5].

```
Definition canon_proj(r:Set):= BL(fun x=> class r x)
  (substrate r) (quotient r).

Lemma function_canon_proj: forall r,
  is_equivalence r -> is_function (canon_proj r).
Lemma W_canon_proj: forall r x,
```

[5]The case where R is the equivalence associated to a correspondence is not considered anymore in Version 3.

```
  is_equivalence r ->
  inc x (substrate r) -> W x (canon_proj r) = class r x.
Lemma related_graph_canon_proj: forall r x y,
  is_equivalence r -> inc x (substrate r) -> inc y (quotient r) ->
  inc (J x y) (graph (canon_proj r)) = inc x y.
Lemma canon_proj_show_surjective:forall r x,
  is_equivalence r-> inc x (quotient r)
  ->W (rep x)(canon_proj r) =x.

Lemma canon_proj_surjective:forall r,
  is_equivalence r-> surjective (canon_proj r).
```

The next lemma says that if $A \subset E$ and $x \subset \bar{A}$ (where $\bar{A}$ is the set of all $\bar{a}$ for $a \in A$) then $x \in E/R$. We then state Criterion 55 [2, p. 115]: $u \overset{R}{\sim} v$ if and only if $\bar{u} = \bar{v}$. The exact Bourbaki statement is "Let R be an equivalence relation on a set E, and let p be the canonical mapping of E onto E/R. Then $R\{\!\{x, y\}\!\} \iff (p(x) = p(y))$". The correct statement would be: $R\{\!\{x, y\}\!\}$ if and only if $x \in E$ and $y \in E$ and $p(x) = p(y)$. The proof is a bit strange. It starts with: "let x and y be elements of E such that $(x, y) \in G$. Then $x \in E$ and $y \in E$; let us show..."

```
Lemma sub_im_canon_proj_quotient: forall r a x,
  is_equivalence r -> sub a (substrate r) ->
  inc x (image_by_fun (canon_proj r) a) ->
  inc x (quotient r).
Lemma related_e_rw: forall r u v,
 is_equivalence r -> (related r u v =
   (inc u (source (canon_proj r)) &
     inc v (source (canon_proj r)) &
     W u (canon_proj r) = W v (canon_proj r))).
```

If we consider the equivalence associated with the equality on a set E then each class is a singleton and E/R is equipotent to E.

```
Lemma class_diagonal: forall x u,
  inc u x -> class (diagonal x) u = singleton u.
Lemma bijective_canon_proj_diagonal: forall x,
  bijective (canon_proj (diagonal x)).
```

Consider now the equivalence associated with pr_1. We consider a product $E \times F$ and the relation $(x, y) \sim (x, z)$ for every x, y and z. This is an equivalence on $E \times F$, and classes are objects of the form $\{x\} \times F$. The function $x \mapsto \{x\} \times F$ is a bijection of E onto $(E \times F)/R$.

```
Definition first_proj_equiv (x y:Bet) :=
  eq_rel_associated(first_proj (product x y)).

Definition first_proj_equivalence (x y :Set) :=
  equivalence_associated (first_proj (product x y)).

Lemma first_proj_equiv_pr: forall x y a b,
  first_proj_equiv x y a b =
  (inc a (product x y) & inc b (product x y) & P a = P b).
Lemma graph_first_proj: forall x y,
  is_graph_of(first_proj_equivalence x y)(first_proj_equiv x y).
Lemma equivalence_first_proj: forall x y,
  is_equivalence (first_proj_equivalence x y).
```

```
Lemma first_proj_equivalence_related: forall x y a b,
  related (first_proj_equivalence x y) a b =
  (inc a (product x y) & inc b (product x y) & P a = P b).
Lemma substrate_first_proj_equivalence: forall x y,
  substrate(first_proj_equivalence x y) = product x y.
Lemma first_proj_equivalence_class:forall x y z,
  nonempty y ->
  is_class (first_proj_equivalence x y) z =
  exists u, inc u x & z = product (singleton u) y.   (* 17 *)
Lemma first_proj_equiv_proj:  forall x y,
  nonempty y->
  bijective (fun_function (fun u => product (singleton u) y)
    x (quotient (first_proj_equivalence x y))).
```

For any equivalence, the quotient is a partition of the substrate.

```
Lemma sub_quotient_powerset: forall r,
  is_equivalence r -> sub (quotient r) (powerset (substrate r)).
Lemma partition_from_equivalence: forall r,
  is_equivalence r ->
  partition(quotient r)(substrate r).
```

We consider now the converse. Let f be a function and F its graph. Assume that F is a partition of X. We shall write X_i instead of $f(i)$. We can consider the relation $x \sim y$ defined by: there is an i such that $x \in X_i$ and $y \in X_i$. This relation has a graph on X, say r. Then r is an equivalence on X. We have $a \in \overline{b}$ (i.e., a and b are related by r) if and only if there is an i such that $a \in X_i$ and $b \in X_i$. If a set a is a class of r, then it has the form X_i. The converse is true if X_i is not empty. In other words, the image of f is the quotient X/r. We restate it as follows: if f is a function and the graph of f is a true partition of X, then $i \mapsto f(i)$ is a bijection from E to X/r.

```
Definition in_same_coset f x y:=
   exists i, inc i (source f) & inc x (W i f) & inc y (W i f).

Definition partition_relation f x :=
  graph_on (in_same_coset f) x.

Lemma reflexive_isc: forall f x, is_function f ->
  partition_fam (graph f) x -> reflexive_r (in_same_coset f) x.
Lemma symmetric_isc: forall f,  symmetric_r (in_same_coset f).
Lemma transitive_isc: forall f x, is_function f ->
  partition_fam (graph f) x -> transitive_r (in_same_coset f).
Lemma equivalence_isc: forall f x, is_function f ->
  partition_fam (graph f) x -> equivalence_re (in_same_coset f) x.
Lemma partition_rel_graph: forall f x,
  is_function f ->  partition_fam (graph f) x ->
  is_graph_of (partition_relation f x) (in_same_coset f).
Lemma partition_relation_pr: forall f x a b,
  is_function f -> partition_fam (graph f) x ->
  related (partition_relation f x) a b = in_same_coset f a b.
Lemma partition_is_equivalence: forall f x,
  is_function f -> partition_fam (graph f) x ->
  is_equivalence (partition_relation f x).
Lemma substrate_partition_relation: forall f x,
  is_function f -> partition_fam (graph f) x ->
```

```
  substrate (partition_relation f x)= x.
Lemma partition_relation_class: forall f x a,
  is_function f -> partition_fam (graph f) x ->
  is_class (partition_relation f x) a
  -> exists u, inc u (source f) & a = W u f.
Lemma partition_relation_class2: forall f x u,
  is_function f -> partition_fam (graph f) x ->
  inc u (source f) -> nonempty (W u f)
  -> is_class (partition_relation f x)  (W u f).
Lemma bijective_partition_fun: forall f x,
  is_function f -> partition_fam (graph f) x ->
  (forall u, inc u (source f) -> nonempty (W u f))
  -> bijective (fun_function (fun u => W u f)
    (source f) (quotient (partition_relation f x))).
```

With the same notations, a system of representatives is a set S such that $X_i \cap S$ is a singleton. The same name is given to an injective function g whose image is a system of representatives. In this case, for every i there is a unique j such that $g(j) \in X_i$. Conversely if this condition holds and g is injective, it is a system of representatives. As a consequence, every right inverse of the canonical projection of X on the quotient set defined by the partition X_i of X is a system of representatives.

```
Definition representative_system s f x :=
   is_function f & partition_fam (graph f) x & sub s x
  & (forall i, inc i (source f) -> is_singleton (intersection2 (W i f) s)).

Definition representative_system_function g f x :=
  injective g & (representative_system (range (graph g)) f x).

Lemma rep_sys_function_pr: forall g f x i,
  representative_system_function g f x -> inc i (source f)
  -> exists_unique (fun a=> inc a (source g) & inc (W a g) (W i f)).
Lemma rep_sys_function_pr2: forall g f x,
  injective g -> is_function f -> partition_fam (graph f) x -> sub (target g) x
  -> (forall i, inc i (source f)
    -> exists_unique (fun a=> inc a (source g) & inc (W a g) (W i f)))
  -> representative_system_function g f x.
Lemma section_canon_proj_pr: forall g f x y r,
  r = partition_relation f x ->  is_function f -> partition_fam (graph f) x
  -> is_right_inverse g (canon_proj r) ->
  inc y x ->
  related r y (W (class r y) g). (* 15 *)
Lemma section_is_representative_system_function: forall g f x,
  is_function f -> partition_fam (graph f) x
  -> is_right_inverse g (canon_proj (partition_relation f x)) ->
  (forall u, inc u (source f) -> nonempty (W u f)) ->
  representative_system_function g f x.    (* 31 *)
```

7.3 Relations compatible with an equivalence relation

We say that P(x) is *compatible* with $\sim$ if P(x) and $x \sim y$ imply P(y). Every property is compatible with the equality.

```
Definition compatible_with_equiv_p (p:EP)(r:Set) :=
```

```
  forall x x', p x -> related r x x' -> p x'.

Lemma trivial_equiv: forall p x,
  compatible_with_equiv_p p (diagonal x).
```

If p is compatible with $\sim$, we can define $P(t)$ on the quotient E/R of $\sim$ by:

$$t \in E/R \text{ and } (\exists x)(x \in t \text{ and } P\{x\})$$

It is said to be *induced* by $p\{x\}$ on passing to the quotient (with respect to x) with respect to R. If there is $x \in t$ with $p(x)$, then for all $x \in t$ we have $p(x)$. This is Criterion C56. If x is in the substrate, then $p(x)$ is equivalent to $P(\bar{x})$ where $\bar{x}$ is the class of x.

```
Definition relation_on_quotient p r :=
  fun t => inc t (quotient r) & exists x, inc x t & p x.

Lemma rel_on_quo_pr: forall  p r t,
  is_equivalence r -> compatible_with_equiv_p p r ->
  relation_on_quotient p r t = (inc t (quotient r) &  forall x, inc x t -> p x).
Lemma rel_on_quo_pr2: forall p r y,
  is_equivalence r -> compatible_with_equiv_p p r ->
  (inc y (substrate r) & relation_on_quotient p r (W y (canon_proj r))) =
  (inc y (substrate r) & p y).
```

7.4 Saturated subsets

A subset A of the substrate of a relation r is said *saturated* if $x \in A$ is compatible with r. This is the same as saying that for every $y \in A$ the class of y is a subset of A, or that there exists a set B formed by classes modulo r whose union is A.

```
Definition saturated(r x:Set) := compatible_with_equiv_p (fun y=> inc y x) r.

Lemma saturated_pr: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  (saturated r x) =  (forall y, inc y x -> sub (class r y) x).
Lemma saturated_pr2: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  (saturated r x) =  exists y, (forall z, inc z y -> is_class r z)
    & x = union y.
```

Given a function f and a set X, we consider X_f to be $f^{-1}\langle f\langle X\rangle\rangle$. We have $y \in X_f$ if and only if there is a $z \in X$ such that $f(y) = f(z)$. If we have an equivalence relation r and f is the canonical projection onto the quotient set, then $f(y) = f(z)$ is the same as $y \overset{r}{\sim} z$. If X is the singleton $\{x\}$, then X_f is the class of x modulo r. As a consequence X is saturated if and only if $X = X_f$. If X is part of the substrate, it is saturated if and only if it is the inverse image (of some set) by the canonical projection on the quotient set.

```
Definition inverse_direct_value f x :=
  image_by_fun (inverse_fun f) (image_by_fun f x).
Lemma class_is_inv_direct_value: forall r x,
  is_equivalence r -> inc x (substrate r) ->
  class r x = inverse_direct_value (canon_proj r) (singleton x).   (* 15 *)
Lemma saturated_pr3: forall r x,
```

```
  is_equivalence r -> sub x (substrate r) ->
  saturated r x =  (x= inverse_direct_value (canon_proj r) x).  (* 21 *)
Lemma saturated_pr4: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  saturated r x =  (exists b, sub b (quotient r)
    & x = image_by_fun (inverse_fun (canon_proj r)) b).
```

The following lemmas show that *saturated* behaves friendly with union, intersection and complement.

```
Lemma saturated_union: forall r x,
  is_equivalence r -> (forall y, inc y x -> sub y (substrate r)) ->
  (forall y, inc y x -> saturated r y) ->
  (sub (union x) (substrate r) &  saturated r (union x)).
Lemma saturated_intersection : forall r x,
  is_equivalence r -> nonempty x ->
  (forall y, inc y x -> sub y (substrate r)) ->
  (forall y, inc y x -> saturated r y) ->
  (sub (intersection x) (substrate r) &  saturated r (intersection x)).
Lemma saturated_complement : forall r a,
  is_equivalence r -> sub a (substrate r) -> saturated r a ->
  saturated r (complement (substrate r) a).
```

The set X_f is called the saturation of X by r if f is the canonical projection associated to r. It is the union of classes of elements of X. It is the smallest saturated set that contains X. If X_i is a family of sets, A_i their saturations, then the saturation of $\bigcup X_i$ is $\bigcup A_i$.

```
Definition saturation_of (r x:Set):=
  inverse_direct_value (canon_proj r) x.
Lemma saturation_of_pr: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  saturation_of r x =
  union (Zo (quotient r)(fun z=> exists i, inc i x & z = class r i)).   (* 16 *)
Lemma saturation_of_smallest: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  (saturated r (saturation_of r x) &
    sub x (saturation_of r x)
    & (forall y, sub y (substrate r) -> saturated r y -> sub x y
      -> sub (saturation_of r x) y)).      (* 18 *)

Definition union_image x f:=
  union (Zo x (fun z=> exists i, inc i (source f) & z = W i f)).

Lemma saturation_of_union: forall r f g,
  is_equivalence r -> is_function f -> is_function g ->
  (forall i, inc i (source f) -> sub (W i f) (substrate r)) ->
  source f = source g ->
  (forall i, inc i (source f) -> saturation_of r (W i f) = W i g)
  -> saturation_of r (union_image (powerset(substrate r)) f) =
  union_image (powerset(substrate r)) g.     (* 20 *)
```

7.5 Mappings compatible with equivalence relations

We start with some properties of the function s that maps a non-empty set x to a representative: If R is an equivalence relation, this is a function from E/R to E; it is a section (right inverse) of the canonical projection.

```
Definition section_canon_proj (r:Set) :=
  fun_function rep (quotient r) (substrate r).
Lemma axioms_section_canon_proj:forall r,
  is_equivalence r ->
  transf_axioms rep (quotient r) (substrate r).
Lemma W_section_canon_proj: forall r x,
  is_equivalence r ->
  inc x (quotient r) -> W x (section_canon_proj r) = (rep x).
Lemma function_section_canon_proj: forall r,
  is_equivalence r -> is_function (section_canon_proj r).
Lemma right_inv_canon_proj: forall r,
  is_equivalence r ->
  is_right_inverse (section_canon_proj r) (canon_proj r).
```

We say that a function f is *compatible* with R if the relation $f(x) = y$ is compatible; by definition this is: if $x \overset{\mathrm{R}}{\sim} x'$ then $f(x) = y$ implies $f(x') = y$. By symmetry, these two relations are equivalent, and we can eliminate y. We first prove that our definition is the same as the original one, then show that this means that the function is constant on equivalence classes. This means that f can be factored through the canonical projection g (see below; we show here $g(x) = g(y)$ implies $f(x) = f(y)$). (see diagram (retraction/section) on page 65).

```
Definition compatible_with_equiv f r :=
  is_function f & source f = substrate r  &
  forall x x', related r x x' -> W x f = W  x' f.

Lemma compatible_with_equiv_pr: forall f r,
  is_function f -> source f = substrate r  ->
  compatible_with_equiv f r =
  (forall y, compatible_with_equiv_p (fun x => y = W x f) r).
Lemma compatible_constant_on_classes: forall f r x y,
  is_equivalence r ->
  compatible_with_equiv f r -> inc y (class r x) -> W x f = W y f.
Lemma compatible_constant_on_classes2: forall f r x,
  is_equivalence r -> compatible_with_equiv f r ->
  is_constant_function(restriction_function f (class r x)).
Lemma compatible_with_proj: forall f r x y,
  is_equivalence r -> compatible_with_equiv f r ->
  inc x (substrate r) -> inc y (substrate r) ->
  W x (canon_proj r) = W y (canon_proj r) -> W x f = W y f.
```

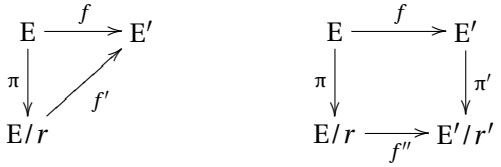
Given two relations r and s, we say that the function f is *compatible* with r and s if $g \circ f$ is compatible with r, when g is the canonical projection of F/s. We can restate this as: $x \overset{r}{\sim} y$ implies $f(x) \overset{s}{\sim} f(y)$. If h is the canonical projection of E/r, then $h(x) = h(y)$ implies that $f(x)$ and $f(y)$ have the same class modulo s.

```
Definition compatible_with_equivs f r r' :=
  is_function f & target f =  substrate r' &
  compatible_with_equiv (compose (canon_proj r') f) r.
```

```
Lemma compatible_with_pr:forall r r' f x y,
  is_equivalence r ->  is_equivalence r' ->
  compatible_with_equivs f r r' ->
  related r x y -> related r' (W x f) (W y f).
Lemma compatible_with_pr2:forall r r' f,
  is_equivalence r ->  is_equivalence r' ->
  is_function f ->
  target f = substrate r'-> source f = substrate r->
  (forall x y,  related r x y -> related r' (W x f) (W y f)) ->
  compatible_with_equivs f r r'.
Lemma compatible_with_proj3 :forall r r' f x y,
  is_equivalence r -> is_equivalence r' ->
  compatible_with_equivs f r r'->
  inc x (substrate r) -> inc y (substrate r) ->
  W x (canon_proj r) = W y (canon_proj r) ->
  class r' (W x f) = class r' (W y f).
```

Assume that f is compatible with an equivalence r on E, let g be the canonical projection onto E/r and s a section of g. If f is compatible with r, there exists a unique function h such that $h \circ g = f$ and $h = f \circ s$. This mapping is said to be *induced by f on passing to the quotient*. This is criterion C57 (for details, see page 195).

```
Definition fun_on_quotient r f :=
  compose f (section_canon_proj r).

Lemma exists_fun_on_quotient: forall f r,
  is_equivalence r -> is_function f -> source f = substrate r ->
  compatible_with_equiv f r =
  (exists h, composable  h (canon_proj r) & compose h (canon_proj r) = f).
Lemma exists_unique_fun_on_quotient: forall f r h,
  is_equivalence r -> compatible_with_equiv f r ->
  composable  h (canon_proj r) -> compose h (canon_proj r) = f ->
  h = fun_on_quotient r f.
Lemma compose_foq_proj :forall f r,
  is_equivalence r ->  compatible_with_equiv f r ->
  compose (fun_on_quotient r f)  (canon_proj r) = f.
```



(fun on quotient)

Assume that f is a function from E into E′ on which we have equivalence relations r and r'. Let π and π' be the canonical projections onto E/r and E′/r', s and s' associated sections. We can consider $f = f \circ s'$, the mapping induced by f on passing on the quotient, or $f'' = \pi \circ f \circ s$, the mapping induced by f on passing to the quotients with respect to r and s. We consider two cases: f is a mapping, and f is a graph. In order to simplify the statements, we write X and X′ instead of *is_equivalence r* or *is_equivalence r'*.

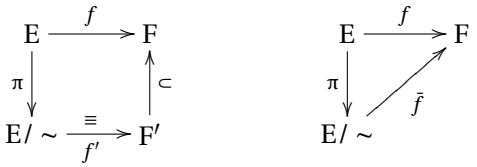
```
Definition fun_on_rep f:EE := fun x=> f(rep x).
Definition fun_on_reps r' f := fun x=> W (f(rep x)) (canon_proj r').
Definition function_on_quotient r f b  :=
  BL(fun_on_rep f)(quotient r)(b).
```

```
Definition function_on_quotients r r' f :=
  BL(fun_on_reps r' f)(quotient r)(quotient r').
Definition fun_on_quotients r r' f :=
  compose (compose (canon_proj r') f) (section_canon_proj r).
Lemma axioms_foq: forall r f b, X->
  transf_axioms f (substrate r) b ->
  transf_axioms (fun_on_rep f) (quotient r) b.
Lemma axioms_foqs: forall r r' f, X -> X' ->
  transf_axioms f (substrate r)(substrate r') ->
  transf_axioms (fun_on_reps r' f) (quotient r) (quotient r').
Lemma axioms_foqc: forall r f, X->
  is_function f -> source f = substrate r ->
  composable f (section_canon_proj r).
Lemma axioms_foqcs:forall r r' f, X-> X'->
  is_function f -> source f = substrate r -> target f = substrate r' ->
  composable (compose (canon_proj r') f) (section_canon_proj r).

Lemma function_foq:forall r f b, X->
  transf_axioms f (substrate r) b ->
  is_function (function_on_quotient r f b).
Lemma function_foqs: forall r r' f, X-> X' ->
  transf_axioms f (substrate r)(substrate r') ->
  is_function (function_on_quotients r r' f).
Lemma function_foqc: forall r f, X-> X' ->
  source f = substrate r ->
  is_function (fun_on_quotient r f).
Lemma function_foqcs:forall r r' f, X-> X' ->
  is_function f -> source f = substrate r -> target f = substrate r' ->
  is_function (fun_on_quotients r r' f).
Lemma W_foq:forall r f b x, X->
  transf_axioms f (substrate r) b ->
  inc x (quotient r) ->
  W x (function_on_quotient r f b) = f (rep x).
Lemma W_foqc:forall r f x, X ->
   is_function f ->
  source f = substrate r -> inc x (quotient r) ->
  W x (fun_on_quotient r f) = W (rep x) f.
Lemma W_foqs: forall r r' f x, X -> X' ->
  transf_axioms f (substrate r)(substrate r') ->   inc x (quotient r) ->
  W x (function_on_quotients r r' f) = class r' (f (rep x)).
Lemma W_foqcs:forall r r' f x, X-> X' ->
  is_function f -> source f = substrate r -> target f = substrate r' ->
  inc x (quotient r) ->
  W x (fun_on_quotients r r' f) = class r' (W (rep x) f).
```

More lemmas; statement *fun_on_quotient_pr4* is the diagram on the right part of (fun on quotient) on page 123.

```
Lemma fun_on_quotient_pr: forall r f x,
  W x f = fun_on_rep (fun w => W x f) (W x (canon_proj r)).
Lemma fun_on_quotient_pr2: forall r r' f x,
  W (W x f) (canon_proj r') =
  fun_on_reps r' (fun w => W x f) (W x (canon_proj r)).
Lemma composable_fun_proj: forall r f b, X->
  transf_axioms f (substrate r) b ->
  composable (function_on_quotient r f b) (canon_proj r).
Lemma composable_fun_projs: forall r r' f, X -> X' ->
```

```
  transf_axioms f (substrate r) (substrate r') ->
  composable (function_on_quotients r r' f) (canon_proj r).
Lemma composable_fun_projc: forall r f, X->
  compatible_with_equiv f r ->
  composable (fun_on_quotient r f) (canon_proj r).
Lemma composable_fun_projcs: forall r r' f, X-> X' ->
  compatible_with_equivs f r r'->
  composable (fun_on_quotients r r' f) (canon_proj r).
Lemma fun_on_quotient_pr3: forall r f x, X->
  inc x (substrate r) ->   compatible_with_equiv f r ->
  W x f = W (W x (canon_proj r)) (fun_on_quotient r f).
Lemma fun_on_quotient_pr4: forall r r' f, X-> X' ->
  compatible_with_equivs f r r'->
  compose (canon_proj r') f = compose (fun_on_quotients r r' f)(canon_proj r).
Lemma fun_on_quotient_pr5: forall r r' f x, X-> X'->
  compatible_with_equivs f r r'->
  inc x (substrate r) ->
  W (W x f) (canon_proj r') =
  W (W x (canon_proj r)) (fun_on_quotients r r' f).
Lemma compose_fun_proj_ev: forall r f b x, X->
  compatible_with_equiv (fun_function f (substrate r) b) r ->
  inc x (substrate r) ->
  transf_axioms f (substrate r) b ->
  W x (compose (function_on_quotient r f b) (canon_proj r)) = f x.
Lemma compose_fun_proj_ev2: forall r r' f x, X-> X' ->
  compatible_with_equivs (fun_function f (substrate r) (substrate r')) r r' ->
  transf_axioms f (substrate r) (substrate r') ->
  inc x (substrate r) ->
  inc (f x) (substrate r') ->
  W (f x) (canon_proj r') =
  W x (compose (function_on_quotients r r' f) (canon_proj r)).
Lemma compose_fun_proj_eq: forall r f b, X->
  compatible_with_equiv (fun_function f (substrate r) b) r ->
  transf_axioms f (substrate r) b ->
  compose (function_on_quotient r f b) (canon_proj r) =
    fun_function f (substrate r) b.
Lemma compose_fun_proj_eq2: forall r r' f, X-> X' ->
  transf_axioms f (substrate r) (substrate r') ->
  compatible_with_equivs (fun_function f (substrate r) (substrate r')) r r'->
  compose (function_on_quotients r r' f) (canon_proj r) =
  compose (canon_proj r') (fun_function f (substrate r) (substrate r')).
```



(canonical decomposition)

Assume now that f is a function from E to F, and $\sim$ the associated equivalence, for which x and y are equivalent if $f(x) = f(y)$. Then f is compatible and we can define f on the quotient. If we denote it by $\bar{f}$, and if $\bar{x}$ is the class of x then $\bar{f}(\bar{x}) = f(x)$. From $\bar{f}(\bar{x}) = \bar{f}(\bar{y})$ we get $f(x) = f(y)$, so that x and y are in the same class: hence $\bar{f}$ is injective. If we restrict this function to the image F' of f we get a bijection, say f'. The diagram (canonical decomposition) says that if we compose the projection π from E to $E/\sim$, the bijection f' into F' and the inclusion map from F' to F, then we get f. If f is surjective then $F = F'$ and we can simplify a

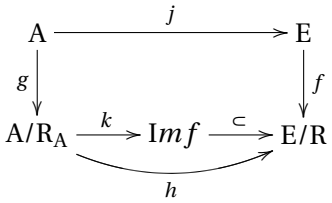
bit: only three arrows are needed. Moreover, there is no need to restrict $\bar{f}$ (this is shown on the right part of the diagram).

```
Lemma compatible_ea: forall f,
  is_function f ->
  compatible_with_equiv f (equivalence_associated f).
Lemma ea_foq_injective: forall f,
  is_function f ->
  injective (fun_on_quotient (equivalence_associated f) f).
Lemma ea_foq_on_im_bijective: forall f,
  is_function f ->
  bijective (restriction2 (fun_on_quotient (equivalence_associated f) f)
    (quotient (equivalence_associated f)) (range (graph f))).   (* 22 *)
Lemma canonical_decompositiona: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    is_function (compose (restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    (canon_proj r)).  (* 17 *)
Lemma canonical_decomposition: forall f,
  is_function f ->
  let r:= equivalence_associated f in
  f = compose (canonical_injection (range (graph f))(target f))
  (compose (restriction2 (fun_on_quotient r f)
    (quotient r) (range (graph f)))
  (canon_proj r)). (* 25 *)
Lemma surjective_pr7:  forall f,
  surjective f ->
  canonical_injection (range (graph f))(target f) = identity_fun (target f).
Lemma canonical_decompositiona: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    is_function (compose (restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    (canon_proj r)).
Lemma canonical_decomposition_surj: forall f,
  surjective f ->
  let r:= equivalence_associated f in
  f =  compose (restriction2 (fun_on_quotient r f) (quotient r) (target f))
      (canon_proj r).
Lemma canonical_decompositionb: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    restriction2 (fun_on_quotient r f)  (quotient r) (target f) =
    (fun_on_quotient r f).
Lemma canonical_decomposition_surj2: forall f,
  surjective f ->
  let r:= equivalence_associated f in
  f =  (compose (fun_on_quotient r f) (canon_proj r)).
```

7.6 Inverse image of an equivalence relation; induced equivalence relation

If ϕ is a function from E to F, S an equivalence on F, and u the canonical projection from F to F/S, the inverse image of S by ϕ is the equivalence R associated to $u \circ \phi$, characterized by $x \overset{R}{\sim} y$ if and only if $\phi(x) \overset{S}{\sim} \phi(y)$. If X is a class modulo S then $\phi^{-1}\langle X\rangle$ is a class modulo R (if nonempty) and conversely.

```
Definition inv_image_relation f r :=
  equivalence_associated (compose (canon_proj r) f).
Definition iirel_axioms f r :=
  is_function f & is_equivalence r & target f = substrate r.

Lemma iirel_function: forall f r,
  iirel_axioms f r -> is_function (compose (canon_proj r) f).
Lemma relation_iirel: forall f r,
  iirel_axioms f r -> is_equivalence (inv_image_relation f r).
Lemma substrate_iirel: forall f r,
  iirel_axioms f r -> substrate (inv_image_relation f r) = source f.
Lemma related_iirel: forall f r x y,
  iirel_axioms f r ->
  related (inv_image_relation f r) x y =
  (inc x (source f) & inc y (source f) &  related r (W x f) (W y f)).
Lemma class_iirel: forall f r x,
  iirel_axioms f r ->
  is_class (inv_image_relation f r) x =
  exists y, is_class r y
    & nonempty (intersection2 y (range (graph f)))
    & x = inv_image_by_fun f y.    (* 30 *)
```



(induced equivalence)

Let R be an equivalence on E, A a subset on E, and j the inclusion map $A \to E$. The inverse image of R by j is called the relation *induced* on A and is denoted by R_A. If x and y are in A, then they are related by R_A if and only if they are related by R. Classes for R_A are nonempty sets of the form $A \cap X$ where X is a class for R. The inclusion map is compatible with the relations. Let f and g be the canonical projections and h the function on the quotient. This function is injective, its range is the range of f. Hence h is the composition of a bijection k with the inclusion map.

```
Definition induced_relation (r a:Set) :=
  inv_image_relation (canonical_injection a (substrate r)) r.
Definition axioms_induced_rel(r a :Set) :=
   is_equivalence r & sub a (substrate r).

Lemma axioms_induced_rel_iirel : forall r a,
  axioms_induced_rel r a ->
  iirel_axioms (canonical_injection a (substrate r)) r.
```

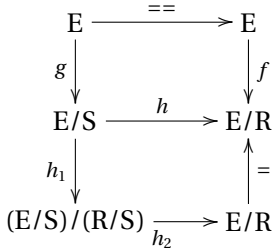
```
Lemma equivalence_induced_rel: forall r a,
  axioms_induced_rel r a -> is_equivalence (induced_relation r a).
Lemma substrate_induced_rel: forall r a,
  axioms_induced_rel r a -> substrate (induced_relation r a) = a.
Lemma related_induced_rel: forall r a u v,
  axioms_induced_rel r a ->
  related (induced_relation r a) u v =
  (inc u a & inc v a & related r u v).
Lemma class_induced_rel: forall r a x,
  axioms_induced_rel r a ->
  is_class (induced_relation r a) x =
  exists y, is_class r y
    & nonempty (intersection2 y a)
    & x = (intersection2 y a).
Lemma compatible_injection_induced_rel: forall r a,
  axioms_induced_rel r a ->
  compatible_with_equivs (canonical_injection a (substrate r))
  (induced_relation r a) r.
Lemma injective_foq_induced_rel: forall r a,
  axioms_induced_rel r a ->
  injective (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r))).
Lemma image_foq_induced_rel: forall r a,
  axioms_induced_rel r a ->
  image_by_fun  (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r))) (quotient (induced_relation r a))
  = image_by_fun (canon_proj r) a.  (* 41 *)
Definition canonical_foq_induced_rel r a :=
  restriction2 (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r)))
  (quotient (induced_relation r a))
  (image_by_fun (canon_proj r) a).
Lemma bijective_canonical_foq_induced_re: forall r a,
  axioms_induced_rel r a -> bijective (canonical_foq_induced_rel r a). (* 15 *)
```

7.7 Quotients of equivalence relations

We say that a relation S is *finer* than R if S implies R. We say that an equivalence r is finer than s if $\overset{s}{\sim}$ implies $\overset{r}{\sim}$, i.e., if for all x and y, $x \overset{s}{\sim} y$ implies $x \overset{r}{\sim} y$. If r and s are equivalences on a same set, this is equivalent to $s \subset r$. If we denote by $C_s x$ the class of x for s, it is also: for each x, there is an y such that $C_s x \subset C_r y$. Equivalently: each $C_r y$ is saturated by s. We give two examples.

```
Definition finer_equivalence(s r:Set):=
  forall x y, related s x y -> related r x y.

Definition finer_axioms(s r:Set):=
  is_equivalence s & is_equivalence r & substrate r = substrate s.

Lemma finer_sub_equiv: forall s r,
  finer_axioms s r ->
  (finer_equivalence s r) = (sub s r).
Lemma finer_sub_equiv2: forall s r,
  finer_axioms s r ->
```

```
  (finer_equivalence s r) =
  (forall x, exists y, sub(class s x)(class r y)).
Lemma finer_sub_equiv3: forall s r,
  finer_axioms s r ->
  (finer_equivalence s r) =
  (forall y, saturated s (class r y)).   (* 15 *)
Lemma finest_equivalence: forall r,
   is_equivalence r -> finer_equivalence (diagonal (substrate r)) r.
Lemma coarsest_equivalence: forall r,
   is_equivalence r -> finer_equivalence r (coarse (substrate r)).
```



(quotient of equivalences)

Assume that R and S are two equivalences on E, S finer than R, and let f and g be the canonical projections. Then f is compatible with S. This gives a surjective function h that satisfies $h(\mathrm{C}_\mathrm{S} x) = \mathrm{C}_\mathrm{R} x$.

```
Lemma compatible_with_finer: forall s r,
  finer_axioms s r ->
  finer_equivalence s r ->
  compatible_with_equiv (canon_proj r) s.
Lemma function_foq_finer: forall s r,
  finer_axioms s r ->
  finer_equivalence s r -> is_function(fun_on_quotient s (canon_proj r)).
Lemma function_foq_finer: forall s r,
  finer_axioms s r ->
  finer_equivalence s r -> is_function(fun_on_quotient s (canon_proj r)).
Lemma W_foq_finer: forall s r x,
  finer_axioms s r -> finer_equivalence s r ->  inc x (quotient s) ->
  W x (fun_on_quotient s (canon_proj r)) = class r (rep x).
Lemma surjective_foq_finer: forall s r,
  finer_axioms s r ->
  finer_equivalence s r -> surjective(fun_on_quotient s (canon_proj r)).
```

On the quotient we can consider the equivalence induced by h. This will be denoted R/S. We have $\mathrm{C}_\mathrm{S} x \overset{\mathrm{R/S}}{\sim} \mathrm{C}_\mathrm{S} y$ if and only if $x \overset{\mathrm{R}}{\sim} y$; this is the same as $g(x) \overset{\mathrm{R/S}}{\sim} g(y)$. We have $x \in (\mathrm{E/S})/(\mathrm{R/S})$ if and only if there exists $y \in \mathrm{E/R}$ such that $y = g(y)$. We can consider the canonical decomposition of $h = j \circ h_2 \circ h_1$. Since h is surjective, we can simplify this as $h = h_2 \circ h_1$; here h_1 is the canonical projection of E/S onto (E/S)/(R/S).

```
Definition quotient_of_relations (r s:Set):=
  equivalence_associated (fun_on_quotient s (canon_proj r)).

Lemma relation_quotient_of_relations:
  forall r s, finer_axioms s r -> finer_equivalence s r ->
    is_equivalence (quotient_of_relations r s).
Lemma substrate_quotient_of_relations:
```

```
  forall r s, finer_axioms s r -> finer_equivalence s r ->
    substrate (quotient_of_relations r s) = (quotient s).
Lemma related_quotient_of_relations: forall r s x y,
  finer_axioms s r -> finer_equivalence s r ->
  related (quotient_of_relations r s) x y =
  (inc x (quotient s) & inc y (quotient s) &
    related r (rep x) (rep y)).
Lemma related_quotient_of_relations_bis: forall r s x y,
  finer_axioms s r -> finer_equivalence s r ->
  inc x (substrate s) -> inc y (substrate s) ->
  related (quotient_of_relations r s) (class s x) (class s y)
  = related r x y.
Lemma nonempty_image: forall f x,
  is_function f -> nonempty x -> sub x (source f) ->
  nonempty (image_by_fun f x).
Lemma cqr_aux: forall s x y u,
  is_equivalence s -> sub y (substrate s) ->
  x = image_by_fun (canon_proj s) y ->
  inc u x = (exists v, inc v y & u = class s v).
Lemma class_quotient_of_relations_bis: forall r s x,
  finer_axioms s r -> finer_equivalence s r ->
  inc x (quotient (quotient_of_relations r s)) =
  exists y, inc y (quotient r) & x = image_by_fun (canon_proj s) y.  (* 59 *)
```

Let S be an equivalence on E and g the canonical projection. Let T be an equivalence on the quotient, and R the inverse image of T by g. This is a relation on E, S is finer than R and R/S is nothing else than T.

```
Lemma quotient_canonical_decomposition: forall r s,
  let f := fun_on_quotient s (canon_proj r) in
    let qr := quotient_of_relations r s in
  finer_axioms s r -> finer_equivalence s r ->
  f =  (compose (fun_on_quotient qr f) (canon_proj qr)).
Lemma quotient_of_relations_pr: forall s t,
  let r := inv_image_relation (canon_proj s) t in
  is_equivalence s -> is_equivalence t -> substrate t = quotient s ->
  t = quotient_of_relations r s.  (* 25 *)
```

7.8 Product of two equivalence relations

Given two relations R and R′, we can define R × R′ by $(x,x') \overset{R\times R'}{\sim} (y,y')$ if and only if $x \overset{R}{\sim} y$ and $x' \overset{R'}{\sim} y'$. In the definition that follows, we consider relations on sets E and E′, and show that this gives a relation on E × E′ (the substrate is not the whole product: if E and E′ have two elements that are related, then the graphs of R and R′ have a single element, the graph of R×R′ has a single element, and its substrate has two elements, while E×E′ has four elements). If R and R′ are equivalence, so is the product, and the substrate is E×E′. A class in the product is a product of classes.

```
Definition substrate_for_prod(r r':Set) :=
  product(substrate r)(substrate r').

Definition prod_of_relation(r r':Set):=
  Zo(product(substrate_for_prod r r')(substrate_for_prod r r'))
```

```
  (fun y=> inc (J(P (P y))(P (Q y))) r & inc (J(Q (P y))(Q (Q y))) r').

Lemma prod_of_rel_is_rel: forall r r', is_graph (prod_of_relation r r').
Lemma substrate_prod_of_rel1:  forall r r',
  sub (substrate (prod_of_relation r r'))(substrate_for_prod r r').
Lemma prod_of_rel_pr: forall r r' a b,
  related (prod_of_relation r r')  a b =
  ( is_pair a & is_pair b & related r (P a) (P b) & related r' (Q a) (Q b)).
Lemma substrate_prod_of_rel2:
  forall r r',  is_symmetric r -> is_symmetric r' ->
    substrate (prod_of_relation r r') = substrate_for_prod r r'.
Lemma prod_of_rel_refl:
  forall r r',  is_reflexive r -> is_reflexive r' ->
   is_reflexive (prod_of_relation r r').
Lemma prod_of_rel_sym:
  forall r r',  is_symmetric r -> is_symmetric r' ->
   is_symmetric (prod_of_relation r r').
Lemma prod_of_rel_trans:
  forall r r',  is_transitive r -> is_transitive r' ->
   is_transitive (prod_of_relation r r').
Lemma substrate_prod_of_rel: forall r r',
  is_equivalence r ->is_equivalence r' ->
  substrate (prod_of_relation r r') =  product(substrate r)(substrate r').
Lemma equivalence_prod_of_rel:  forall r r',
  is_equivalence r -> is_equivalence r' ->
  is_equivalence (prod_of_relation r r').
Lemma related_prod_of_rel1: forall r r' x x' v,
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  related (prod_of_relation r r') (J x x') v =
  (exists y, exists y', v = J y y' & related r x y & related r' x' y').
Lemma related_prod_of_rel2: forall r r' x x' v,
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  related (prod_of_relation r r') (J x x') v =
  inc v (product (im_singleton r x) (im_singleton r' x')).
Lemma class_prod_of_rel2: forall r r' x,
  is_equivalence r -> is_equivalence r' ->
  is_class (prod_of_relation r r') x =
  exists u, exists v, is_class r u & is_class r' v & x = product u v.   (* 21 *)
```

With the same notations, let π and π' be the canonical projections. We can consider the function $\pi \times \pi'$, it maps (x, y) to $(\pi(x), \pi'(x))$: its target is $(E/R) \times (E/R')$. This function is not the canonical projection π'' associated to $R \times R'$, whose target is $(E \times E)/(R \times R')$. However there is a bijection h such that $\pi \times \pi' = h \circ \pi''$.

```
Lemma function_ext_to_prod_rel: forall r r',
  is_equivalence r -> is_equivalence r' ->
  is_function (ext_to_prod(canon_proj r)(canon_proj r')).
Lemma W_ext_to_prod_rel: forall r r' x x',
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  W (J x x') (ext_to_prod(canon_proj r)(canon_proj r')) =
  J (class r x) (class r' x').
Lemma compatible_ext_to_prod: forall r r',
  is_equivalence r -> is_equivalence r' ->
```

```
   compatible_with_equiv (ext_to_prod (canon_proj r) (canon_proj r'))
     (prod_of_relation r r').
Lemma compatible_ext_to_prod_inv: forall r r' x x',
  is_equivalence r -> is_equivalence r' ->
  is_pair x -> inc (P x) (substrate r) -> inc (Q x)  (substrate r') ->
  is_pair x' -> inc (P x') (substrate r) -> inc (Q x')  (substrate r') ->
  W x (ext_to_prod (canon_proj r) (canon_proj r')) =
  W x' (ext_to_prod (canon_proj r) (canon_proj r'))
  -> related (prod_of_relation r r') x x'.
Lemma related_ext_to_prod_rel: forall r r',
  is_equivalence r -> is_equivalence r' ->
  equivalence_associated (ext_to_prod(canon_proj r)(canon_proj r')) =
  prod_of_relation r r'.    (* 31 *)
Lemma decomposable_ext_to_prod_rel:forall r r',
  is_equivalence r -> is_equivalence r' ->
  exists h,  bijective h &
    source h = quotient (prod_of_relation r r') &
    target h = product (quotient r) (quotient r')&
    compose h (canon_proj (prod_of_relation r r')) =
    ext_to_prod(canon_proj r)(canon_proj r').   (* 47 *)
```

7.9 Classes of equivalent objects

Let $\sim$ be an equivalence relation; we do not assume that it has a graph. Let θx be the generic object associated to x. In Bourbaki's notation, this is $\tau_y(x \sim y)$. We could implement this via *chooseT*. Assume $x \sim x'$. Then $x \sim y$ and $x' \sim y$ are equivalent, and the properties of τ say $\theta x = \theta x'$. The quantity θx is the class of objects equivalent to x. Bourbaki notes that "$x \sim x$ and $x' \sim x'$ and $\theta x = \theta x'$" is equivalent to $x \sim x'$.

Assume now that there is a set T such that $y \sim y$ implies that there exists $x \in \mathrm{T}$ such that $x \sim y$. Let Θ be the set of all θx for $x \in \mathrm{T}$. If $y \sim y$, there exists $x \in \mathrm{T}$ such that $x \sim y$, hence $\theta x = \theta y$ and thus $\theta y \in \Theta$. If $x \sim x$, then θx is the unique $z \in \Theta$ such that $x \sim z$.

Assume that $x \sim y$ implies $\mathrm{A}x = \mathrm{A}y$. We can consider the set of all $\mathrm{A}x$ such that $x \sim x$. If f maps t to $\mathrm{A}t$, then we have $\mathrm{A}x = f(\theta x)$. Bourbaki says that if we have an equivalence relation on a set E, then we can choose for $\mathrm{A}x$ the class of x, and f becomes a bijection from Θ into the quotient set.

We write θx and $\mathrm{A}x$ instead of $\theta(x)$ and $\mathrm{A}(x)$ in order to emphasize the fact that these objects are not functions. However, θx is a set. No code is associated to this section. It seems that this section is not used in the remaining of the work of Bourbaki; for instance, if we consider the relation X is equipotent to Y, then θX is the cardinal of X. Bourbaki proves the existence of the cardinal by repeating the arguments previously exposed in this section.

Chapter 8

Exercises

We start with some properties of the Theory of Sets, not used elsewhere. We show that $\mathrm{Coll}_y(y \notin y)$ is false. This implies that there is no set x such that for all y we have $y \in x$, but on the contrary, there is a set x such that for no y we have $y \in x$ (this being the empty set). Then we show that for every property p, we have $p(x)$, provided that $x \in \emptyset$.

```
Lemma not_collectivizing_notin:
  ~ (exists z, forall y, inc y z = not (inc y y)).
Proof. red. ir. nin H.
  assert (~ inc x x). red. ir.
  cp H0. rwi (H x) H0. contradiction.
  cp H0. wri (H x) H0. contradiction.
Qed.
Lemma collectivizing_special :
  (exists x, forall y, ~ (inc y x)) &  ~ (exists x, forall y, inc y x).
Proof. split. exists emptyset. ir. red. app emptyset_pr. red. ir. nin H.
  app not_collectivizing_notin.
  exists (Zo x (fun z => ~ (inc z z))). ir. app iff_eq. ir.
  Ztac. am. ir. Ztac.
Qed.
Lemma emptyset_pra: forall x (p: EP),
  inc x emptyset -> (p x).
Proof. ir.  elim (emptyset_pr H).
Qed.
```

8.1 Section 1

1. *Show that the relation* $(x = y) \iff (\forall X)((x \in X) \implies (y \in X))$ *is a theorem.*

Comment. In Bourbaki, you can prove $x = x$ (this is the first theorem) or $(\forall x)(x = x)$ (this is different theorem). In Coq, we can prove only the second property. We try to be as close as possible to the Bourbaki statement by using a section. The quantifiers are still present, but invisible. This looks like the axiom of extend for the relation $\ni$; implication $\Rightarrow$ is trivial; implication $\Leftarrow$ is a consequence a weaker property, where we restrict X to be a singleton, which reads then: $(\forall z)((x = z) \implies (y = z))$.

```
Section exercise1_1.
Variable x y:Set.
```

```
Lemma exercise1_1: (x=y) = (forall X, inc x X -> inc y X).
Proof. ir. ap iff_eq; ir. ue.
  sy. ap singleton_eq. ap H. fprops.
Qed.

End exercise1_1.
```

2. *Show that* $\emptyset \neq \{x\}$ *is a theorem. Deduce that* $(\exists x)(\exists y)(x \neq y)$ *is a theorem.*

Comment: The first claim is really $(\forall x)(\emptyset \neq \{x\})$. Note that the "axiom of the singleton" (for each x there is a set that has a unique element, namely x) asserts that the number of sets is not finite.

```
Lemma exercise1_2: exists x:Set, exists y:Set, x <> y.
Proof. assert (forall x:Set, emptyset <> singleton x). red. ir.
  elim (emptyset_pr (x:=x)). ue.
  exists emptyset. exists (singleton emptyset). ap H. Qed.
```

3. *Let* A *and* B *be two subsets of a set* X. *Show that the relation* $B \subset \complement A$ *is equivalent to* $A \subset \complement B$ *and that the relation* $\complement B \subset A$ *is equivalent to* $\complement A \subset B$.

Comment: The notation $\complement A$ is an abuse of language for $X - A$. One part of the proof needs the law of excluded middle.

We prove first one implication, the other is obtained by symmetry. The second claim follows from the double complement rule.

```
Lemma exercise1_3: forall A B X, sub A X -> sub B X ->
  (sub (complement X B) A = sub (complement X A) B     &
  sub B (complement X A) = sub A (complement X B)).
Proof. ir. assert (Ha:forall a b,  sub a X -> sub b X ->
    sub (complement X b) a -> sub (complement X a) b).
    ir. red. ir. srwi H4. nin (inc_or_not x b). am.
  nin H4. elim H6. app H3. srw. au.
  assert (Hb: forall a b,  sub a X -> sub b X ->
    (sub (complement X b) a = sub (complement X a) b)). ir. ap iff_eq; app Ha.
  split. app Hb.
  cp (Hb _ _ (sub_complement (a:=X) (b:= A))(sub_complement (a:=X) (b:= B))).
  do 2 rwi double_complement H1;am.
Qed.
```

4. *Prove that the relation* $X \subset \{x\}$ *is equivalent to "*$X = \{x\}$ *or* $X = \emptyset$*".*

Comment: The relation $X \subset \{x\}$ is equivalent to $a \in X \iff a = x$. This needs the law of excluded middle; in other terms, from the proof of $X \subset \{x\}$, it is not possible to deduce which alternative, $X = \{x\}$ or $X = \emptyset$ is true.

```
Lemma exercise1_4: forall a b,
  sub a (singleton b) = (a = singleton b \/ a = emptyset).
Proof. ir. ap iff_eq. ir.
  nin (p_or_not_p (sub (singleton b) a)); ir;
    [ left; app extensionality | right ].
  app is_emptyset. red. ir. elim H0. red. ir.
  rw (singleton_eq H2). wrr (singleton_eq (H _ H1)).
```

```
  ir. nin H; rw H; [ ap sub_refl | ap sub_emptyset_any].
Qed.
```

5. *Prove that* $\emptyset = \tau_X(\tau_x(x \in X) \notin X)$.

Comment: We shall give a proof that uses *choose*, which not exactly the same as Bourbaki's τ function. Hence we start, informally, with a Bourbaki proof. We have to show

$$\tau_X(\neg(\exists x)(\neg(x \notin X))) = \tau_X(\neg(\exists x)(x \in X)),$$

(by definition of $\emptyset$, $\forall$ and $\exists$). Write this as $\tau_X(\neg(\exists x)P) = \tau_X(\neg(\exists x)Q)$. According to Scheme S7, it suffices to prove $(\forall X)(\neg(\exists x)P \iff \neg(\exists x)Q)$. Fix X. Criterion C24 says that $\neg x \notin X$ is equivalent to $x \in X$, i.e., $P \iff Q$. From Criterion C31 it follows that $(\exists x)P \iff (\exists x)Q$. Criterion C23 implies $\neg(\exists x)P \iff \neg(\exists x)q$. Qed.

The expression $\tau_x(x \in X)$ is denoted by *rep* in Coq. Write this as R(X). It satisfies the following property: if for some x we have $x \in X$, then $R(X) \in X$. Denote by Y the set $\tau_X(R(X) \notin X)$. It satisfies $R(Y) \notin Y$ (since the empty set satisfies this property). Now $y \in Y$ becomes absurd, so Y is empty.

```
Lemma  exercise1_5:
  emptyset = choose (fun X => ~ (inc (rep X) X)).
Proof. set (p:= fun X => ~ (inc (rep X) X)).
  sy. ap is_emptyset. red. ir.
  assert (p (choose p)). ap choose_pr. uf p. exists emptyset. red. ir.
  elim (emptyset_pr H0). set (X:=choose p) in *. ufi p H0. elim H0.
  uf rep. app choose_pr. exists y. am.
Qed.
```

6. *Consider* $(\forall y)(y = \tau_x((\forall z)(z \in x \iff z \in y)))$. *Show that this axiom* A1′ *implies the axiom of extent* A1.

Assume that A and B are two sets such that $A \subset B$ and $B \subset A$; said otherwise $z \in A \iff z \in B$. The axioms say $A = \tau_x P$ and $B = \tau_x Q$ for some P and Q. Scheme S7 says A = B if P equivalent to Q, which is true by transitivity of equivalence.

In the proof that follows, we must show that $x \mapsto P(x)$ and $x \mapsto Q(x)$ are the same functions, this requires an axiom.

```
Lemma exercise1_6:
  (forall y, y = choose (fun x => (forall z, (inc z x)=(inc z y))))
  -> (forall a b : Set, sub a b -> sub b a -> a = b).
Proof. ir. rw (H a). rw (H b).
  assert ((fun x : Set => forall z : Set, inc z x = inc z a) =
    (fun x : Set => forall z : Set, inc z x = inc z b)).
  app arrow_extensionality. ir.
  ap iff_eq. ir. ap iff_eq. ir. app H0. wrr H2. ir. rw H2. app H1.
  ir. ap iff_eq. ir. app H1. wrr H2. ir. rw H2. app H0. rww H2.
Qed.
```

8.2 Section 2

1. *Let $R\{x,y\}$ be a relation, the letters x and y being distinct; let z be a letter distinct from x and y which does not appear in $R\{x,y\}$. Show that the relation $(\exists x)(\exists y)R\{x,y\}$ is equivalent to*

$$(\exists z)(z \text{ is an ordered pair and } R\{\mathrm{pr}_1 z, \mathrm{pr}_2 z\})$$

and the relation $(\forall x)(\forall y)R\{x,y\}$ is equivalent to

$$(\forall z)(z \text{ is an ordered pair } \Longrightarrow (R\{\mathrm{pr}_1 z, \mathrm{pr}_2 z\})).$$

Comment. Compare this with the section "Function of two variables".

```
Lemma exercise2_1: forall R:EEP,
( (exists x, exists y, R x y) = (exists z, is_pair z & R(P z) (Q z)) &
  (forall x, forall y, R x y) = (forall z, is_pair z -> R(P z) (Q z))).
Proof. ir. split; ap iff_eq; ir.
  nin H; nin H. exists (J x x0). split. fprops. aw.
  destruct H as [x [H1 H2]]. exists (P x). exists (Q x). am.
  ap H.
  assert (is_pair (J x y)). fprops. cp (H _ H0). awi H1. am.
Qed.
```

2. *(a) Show that the relation $\{\{x\},\{x,y\}\} = \{\{x'\},\{x',y'\}\}$ is equivalent to $x = x'$ and $y = y'$.*
(b) Let $\mathcal{T}_0$ be the theory of sets, and let $\mathcal{T}_1$ be the theory which has the same schemes and explicit axioms as $\mathcal{T}_0$, except for the axiom A3. Show that if $\mathcal{T}_1$ is not contradictory, then $\mathcal{T}_0$ is not contradictory.

Comment. In the French version, Bourbaki defines the pair (x,y) as $\{\{x\},\{x,y\}\}$ and proves (a) as Proposition 1 (thus Propositions in this section are numbered differently in the two editions). In the English version, there is a specific sign (that looks a bit like $\supset$) that defines a pair, and an axiom A3. Part (b) of the exercise is then: if the French version is not contradictory, then the English version is neither.

```
Definition xpair (x y : Set) :=
  doubleton (singleton x) (doubleton x (singleton y)).

Lemma exercise2_2 : forall x y z w,
  (xpair x y = xpair z w) = (x = z & y = w).
Proof. ir. ap iff_eq; ir.
  assert (inc (singleton x) (xpair z w)). wr H. uf xpair. fprops.
  assert (inc (doubleton x (singleton y)) (xpair z w)). wr H. uf xpair. fprops.
  assert (x=z). nin (doubleton_or H0). app singleton_inj.
  sy. app singleton_eq. ue.
  split. am.
  wri H2 H1. nin (doubleton_or H1).
  assert (singleton y = x).  ap singleton_eq. ue.
  assert (inc (doubleton x (singleton w)) (xpair x y)).
  rw H. rw H2. uf xpair. fprops. ufi xpair H5. rwi H4 H5.
  do 2 rwi doubleton_singleton H5. cp (singleton_eq H5).
  assert (singleton w = x). app singleton_eq. ue.
  app singleton_inj. ue.
  assert (inc (singleton w)  (doubleton x (singleton y))). ue.
```

```
  induction (doubleton_or H4).
  assert (inc (singleton y)  (doubleton x (singleton w))). ue.
  induction (doubleton_or H6).
  wri H5 H7; app singleton_inj. app singleton_inj.  sy;app singleton_inj.
  nin H. ue. ue.
Qed.
```

8.3 Section 3

1. *Show that the relations $x \in y$, $x \subset y$, $x = \{y\}$ have no graph with respect to x and y.*

We show here a stronger result: there is no set G that contains all pairs (x, y) of related elements.

```
(* Definition has_no_graph (r:EEP):=
   ~(exists G, is_graph G & forall x y, r x y <-> inc (J x y) G). *)
Definition has_no_graph (r:EEP):=
 ~(exists G, forall x y, r x y -> inc (J x y) G).
Definition is_universal (r:EEP):= forall x, exists y, r x y.
Definition is_universal1 (r:EEP):= forall y, exists x, r x y.
```

Assume that r is a universal relation with a graph x. Let D be the domain of x and $E = \{z \in D, z \notin z\}$. Let C be the claim $E \in E$. It implies $E \notin E$, contradiction, so that C is false and $E \notin E$ is true, from which we deduce $E \in E$, a contradiction, since universality of r implies $z \in D$ for any z.

```
Lemma is_universal_pr: forall r,  is_universal r -> has_no_graph r.
Proof. ir. red. red. ir. destruct H0 as [x [H0 H1]].
  set (E:=Zo(domain x)(fun z => ~ (inc z z))).
  assert (~(inc E E)). red. ir. ufi E H2. Ztac. contradiction.
  elim H2. uf E. Ztac. aw.  nin (H E). exists x0. nin (H1 E x0). app H4.
Qed.

Lemma is_universal1_pr: forall r,  is_universal1 r -> has_no_graph r.
Proof. ir. red. red. ir. destruct H0 as [x H0].
  set (E:=Zo(range x)(fun z => ~ (inc z z))).
  assert (~(inc E E)). red. ir. ufi E H1. Ztac. contradiction.
  elim H1. uf E. Ztac. fold E. uf range. aw. nin (H E).
  exists (J x0 E).  split. app H0. aw.
Qed.
```

The result is now trivial.

```
Lemma exercise3_1:
  has_no_graph (fun x y => inc x y) &
  has_no_graph (fun x y => sub x y) &
  has_no_graph (fun x y => x = singleton y).
Proof. ir. split. app is_universal_pr. red. ir. exists (singleton x). fprops.
  split. app is_universal_pr. red. ir.   exists x. fprops.
  app is_universal1_pr. red. ir.  exists (singleton y). tv.
Qed.
```

2. *Let* G *be a graph. Show that the relation* $X \subset \mathrm{pr}_1 G$ *is equivalent to* $X \subset G^{-1}\langle G\langle X\rangle\rangle$.

```
Lemma exercise3_2: forall G X, is_graph G ->
  sub X (domain G) =
  sub X (image_by_graph (inverse_graph G) (image_by_graph G X)).
Proof. ir. assert (Ha:is_graph (inverse_graph G)).  fprops.
  ap iff_eq. ir. red. ir. pose (H0 _ H1).
  awi i. induction i. aw. exists x0. split. aw. ex_tac. aw. am.
  ir. red. ir. pose (H0 _ H1). awi i. induction i. induction H2.
  awi H3; tv. ex_tac.
Qed.
```

3. *Let* G, H *be two graphs. Show that the relation* $\mathrm{pr}_1 H \subset \mathrm{pr}_1 G$ *is equivalent to* $H \subset H \circ G^{-1} \circ G$. *Deduce that* $G \subset G \circ G^{-1} \circ G$.

```
Lemma exercise3_3a: forall G H, is_graph G -> is_graph H ->
  sub (domain H) (domain G)=
  sub H (compose_graph H (compose_graph (inverse_graph G) G)).
Proof. ir. cp (inverse_graph_is_graph (r:= G)).
  assert (is_graph (compose_graph (inverse_graph G) G)).
  app composition_is_graph. app iff_eq. ir.  red. ir.
  assert (J (P x)(Q x)=x).  aw. app H1. wri H6 H5.
  assert (inc (P x) (domain G)). app H4. ex_tac.
  awi H7. induction H7.
  aw. split. am. exists (P x). split. aw. split. fprops. kex_tac. aw. am
  ir. red. ir. awi H5. nin H5. cp (H4 _ H5). awi H6. nin H6. nin H7. nin H7.
  awi H7. nin H7. nin H9. nin H9. ex_tac. am.
Qed.

Lemma exercise3_3b: forall G, is_graph G ->
  sub G (compose_graph G (compose_graph (inverse_graph G) G)).
Proof. ir. wr (exercise3_3a H H).  fprops. Qed.
```

4. *If* G *is a graph show that* $\emptyset \circ G = G \circ \emptyset = \emptyset$ *and that* $G^{-1} \circ G = \emptyset$ *if and only if* $G = \emptyset$.

```
Lemma exercise3_4: forall G, is_graph G ->
  (compose_graph G emptyset = emptyset &
   compose_graph emptyset G = emptyset &
   (compose_graph (inverse_graph G) G = emptyset) = (G = emptyset)).
Proof. ir.
  split. app is_emptyset. red. ir. awi H0. destruct H0 as [_ [z [H1 _ ]]].
  elim (emptyset_pr H1).
  split. app is_emptyset. red. ir. awi H0. destruct H0 as [_ [z [_ H1 ]]].
  elim (emptyset_pr H1).
  app iff_eq. ir. app is_emptyset. red. ir.
  assert (inc (J (P y) (P y)) emptyset). wr H0. aw. split. fprops.
  exists (Q y). split. aw. app H. aw. app H. fprops. elim (emptyset_pr H2).
  ir. rw H0. app is_emptyset. ir. red. ir. awi H1.
  destruct H1 as [_ [z [_ H1 ]]]. elim (emptyset_pr H1). am. fprops.
Qed.
```

5. *Let* A, B *be two sets,* G *a graph.*
Show that $(A \times B) \circ G = G^{-1}\langle A\rangle \times B$ *and* $G \circ (A \times B) = A \times G\langle B\rangle$.

Comment. The code presented here is simpler than in Version 1; we do not need to show that some quantities are graphs.

```
Lemma exercise3_5: forall G A B,
  (compose_graph (product A B) G = product (inv_image_by_graph G A) B &
  compose_graph G (product A B)  = product A (image_by_graph G B)).
Proof. ir.
  split; set_extens; awii H.
  destruct H as [H1 [y [H2 H3]]]. awi H3. ee. aw. ee. am. ex_tac. am.
  destruct H as [H1 [ [y [H2 H3]] H4]]. aw. ee; tv.  ex_tac. fprops.
  destruct H as [H1 [y [H2 H3]]]. awi H2. aw. ee;tv. ex_tac.
  destruct H as [H1 [ H2 [y [H3 H4]]]]. aw. ee. am. ex_tac. fprops.
Qed.
```

6. *For each graph* G *let* G′ *be the graph* $(\mathrm{pr}_1 G \times \mathrm{pr}_2 G) - G$. *Show that* $(G^{-1})' = (G')^{-1}$, *and that* $G \circ (G^{-1})' \subset \Delta'_B$, $(G^{-1})' \circ G \subset \Delta'_A$, *if* $A \supset \mathrm{pr}_1 G$ *and* $B \supset \mathrm{pr}_2 G$. *Show that* $G = (\mathrm{pr}_1 G) \times (\mathrm{pr}_2 G)$ *if and only if* $G \circ (G^{-1})' \circ G = \emptyset$.

```
Definition complement_graph G :=
  complement(product (domain G)(range G)) G.

Lemma complement_graph_g : forall G, is_graph (complement_graph G).
Proof. red. uf complement_graph.  ir. srwi H. nin H.
  rwi inc_product H. nin H; am. Qed.

Lemma exercise3_6a: forall G, is_graph G ->
  complement_graph (inverse_graph G) = inverse_graph(complement_graph G).
Proof. ir. assert (Ha: is_graph (complement_graph G)). ap complement_graph_g.
  uf complement_graph. bw. rww range_inverse.
  set (u:= range G) in *. set (v:= domain G) in *. set_extens.
  srwi H0. nin H0.  awi H0. rw inverse_graph_pr. ee.  am. srw. ee. fprops.
  red. ir. app H1. rw inverse_graph_pr. au. am. am.
  rwi inverse_graph_pr H0. nin H0. srwi H1. nin H1. awii H1. ee.
  srw. split. aw;ee;am. rw inverse_graph_pr. red. ir. nin H5. contradiction.
  am. am.
Qed.

Lemma exercise3_6b: forall G B, is_graph G -> sub (range G) B ->
  sub (compose_graph G (complement_graph (inverse_graph G)))
      (complement_graph (diagonal B)).
Proof. ir. rww  exercise3_6a. red. uf complement_graph. ir. srw.
  rw domain_diagonal. rw range_diagonal. rw inc_diagonal.
  awi H1. destruct H1 as [H2 [y [ H3 H4]]]. awi H3. srwi H3. nin H3.
  split. aw.  ee. am. app H0. set (u:=range G) in *. awi H1. ee; am. am.
  app H0. ex_tac. red. ir. ee. ap H3. ue.
Qed.

Lemma exercise3_6c: forall A G, is_graph G -> sub (domain G) A ->
  sub (compose_graph (complement_graph (inverse_graph G)) G)
      (complement_graph (diagonal A)).
Proof. ir. red. rww exercise3_6a. uf complement_graph.
  rw domain_diagonal. rw range_diagonal. ir. srw. rw inc_diagonal.
  awi H1.  destruct H1 as [H2 [y [ H3 H4]]]. awi H4. srwi H4. nin H4.
  split. aw. ee. am. app H0. ex_tac. app H0.
  set (u:=domain G) in *. awi H1. ee. am. am. red. ir. ee. ap H4. ue.
Qed.

Lemma exercise3_6d: forall G, is_graph G ->
  ( G = product (domain G)(range G) ) =
```

```
  (compose_graph G (compose_graph (complement_graph (inverse_graph G)) G)
  = emptyset ).
Proof. ir.  rw (exercise3_6a H). set (K:= complement_graph G).
  assert(Ha: (G = product (domain G) (range G)) = (K = emptyset)).
  uf K. uf complement_graph. ap iff_eq. ir. wr H0. app complement_itself.
  ir. ap extensionality. app sub_graph_prod. app empty_complement.
  rw Ha.
  app iff_eq. ir. rw H0. rw inverse_graph_emptyset.
  cp (exercise3_4 H). ee. ue.
  ir. ap is_emptyset. red. ir. cp H1.
  ufi K H1. ufi complement_graph H1. srwi H1. nin H1. awii H1. ee.
  nin H4; nin H5.
  elim (emptyset_pr (x:=(J x0 x))).  wr H0. aw. split. fprops.
  ex_tac. aw. split. fprops. ex_tac. aw.
Qed.
```

7. *A graph* G *is functional if and only if for each set* X *we have* $G\langle G^{-1}\langle X\rangle\rangle \subset X$.

```
Lemma exercise3_7: forall g, is_graph g ->
  fgraph g = (forall x, sub (image_by_graph g (inv_image_by_graph g x)) x).
Proof. ir.
  uf inv_image_by_graph. ap iff_eq. ir. red. ir. awi H1. nin H1.
  nin H1. awi H1.  nin H1. nin H1. awi H3.
  red in H0. nin H0. assert (P (J x0 x1) = P (J x0 x)). aw.
  cp (H4 _ _ H3 H2 H5).  wrr (pr2_injective H6).
  ir. red. split. am. ir. cp (H _ H1). cp (H _ H2).
  app pair_extensionality.
  app singleton_eq. app (H0 (singleton (Q y))). aw. exists (P x). split.
  aw. exists (Q y).  split.  fprops. aw. rw H3. aw. red.  aw.
Qed.
```

8. *Let* A, B *be two sets, let* Γ *be a correspondence between* A *and* B*, and let* Γ' *be a correspondence between* B *and* A*. Show that if* $\Gamma'(\Gamma(x)) = \{x\}$ *for all* $x \in A$ *and* $\Gamma(\Gamma'(y)) = \{y\}$ *for all* $y \in B$*, then* Γ *is a bijection of* A *onto* B *and* Γ' *is the inverse mapping.*

Comment. There is an abuse of notation here (see exercise 11). In some cases $\Gamma(x)$ denotes $\Gamma\langle\{x\}\rangle$ and sometimes $\Gamma(X)$ denotes $\Gamma\langle X\rangle$. The proof is a bit longish. In the comments, G and G′ are the graphs.

```
Lemma exercise3_8: forall G G', is_correspondence G -> is_correspondence G' ->
  source G = target G' -> source G' = target G ->
  (forall x, inc x (source G) -> image_by_fun G' (image_by_fun G (singleton x))
    = singleton x) ->
  (forall x, inc x (source G') -> image_by_fun G (image_by_fun G' (singleton x))
    = singleton x) ->
  (bijective G & bijective G' & G = inverse_fun G').
Proof. ir. ufi image_by_fun H3. ufi image_by_fun H4.
  assert (Ha:is_graph (graph G)). fprops.
  assert (Hb:is_graph (graph G')). fprops.
```

If $x \in A$ then x is in the domain of G (since $\Gamma'(\Gamma(x))$ is not empty). Same with G and G′ exchanged.

```
  assert (Hc:source G = domain (graph G)). app extensionality. red. ir.
```

```
cp (H3 _ H5). assert (inc x (singleton x)). fprops. wri H6 H7.
awi H7. nin H7. nin H7. awi H7. nin H7. nin H7. awi H7. rwi H7 H9.
set (aux:=graph G). ex_tac. fprops.
assert (Hd:source G' = domain (graph G')). app extensionality. red. ir.
cp (H4 _ H5). assert (inc x (singleton x)). fprops. wri H6 H7.
awi H7. nin H7. nin H7. awi H7. nin H7. nin H7.   set (aux:=graph G').  aw.
wr (singleton_eq H7). ex_tac. fprops.
```

We show $(x, y) \in \mathrm{G}$ and $(y, z) \in \mathrm{G}'$ implies $x = z$; same with G and G′ exchanged.

```
assert (forall x y z, inc (J x y)(graph G) -> inc (J y z)(graph G') -> x = z).
ir.  assert (inc x (source G)). rw Hc.   set (aux:=graph G). ex_tac.
sy. ap singleton_eq. wr (H3 _ H7). aw. ex_tac. aw. ex_tac. fprops.
assert (forall x y z, inc (J x y)(graph G') -> inc (J y z)(graph G) -> x = z).
ir.  assert (inc x (source G')). rw Hd.   set (aux:=graph G'). ex_tac.
sy. ap singleton_eq. wr (H4 _ H8). aw. ex_tac. aw. ex_tac. frops.
```

We show: if $x \in \mathrm{A}$ there is an y such that $(x, y) \in \mathrm{G}$ and $(y, x) \in \mathrm{G}'$.

```
assert (forall x, inc x (source G) -> exists y,
  inc (J x y) (graph G) & inc (J y x) (graph G')).
ir. cp (H3 _ H7). assert (inc x (singleton x)). fprops. wri H8 H9.
awi H9.  nin H9. nin H9.  awi H9.  nin H9. nin H9.
rwi (singleton_eq H9) H11. ex_tac.
assert (forall x, inc x (source G') -> exists y,
  inc (J x y) (graph G') & inc (J y x) (graph G)).
ir. cp (H4 _ H8). assert (inc x (singleton x)). fprops. wri H9 H10.
awi H10.  nin H10. nin H10.  awi H10.  nin H10. nin H10.
rwi (singleton_eq H10) H12. ex_tac.
```

We show $(x, y) \in \mathrm{G}$ and $(x, z) \in \mathrm{G}$ implies $y = z$.

```
assert (fgraph (graph G)). red. split. am. ir.
assert (is_pair x). app Ha. assert (J (P x) (Q x) = x). aw.
wri H13 H9. assert (inc (P x) (source G)). rw Hc.
set (aux:=graph G). ex_tac.
cp (H7 _ H14). nin H15. nin H15. cp  (H6 _ _ _ H16 H9).
assert (is_pair y). app Ha. assert (J (P y) (Q y) = y). aw.
wri H19 H10. wri H11 H10. cp  (H6 _ _ _ H16 H10).
wr H13; wr H19; wr H17; wr H20; wr H11; tv.
assert (fgraph (graph G')). red. split. am. ir.
assert (is_pair x). app Hb. assert (J (P x) (Q x) = x). aw.
wri H14 H10. assert (inc (P x) (source G')). rw Hd.
set (aux:=graph G').  ex_tac.
cp (H8 _ H15). nin H16. nin H16. cp  (H5 _ _ _ H17 H10).
assert (is_pair y). app Hb. assert (J (P y) (Q y) = y). aw.
wri H20 H11. wri H12 H11. cp  (H5 _ _ _ H17 H11).
wr H14; wr H20; wr H18; wr H21; wr H12; tv.
```

We show $(x, y) \in \mathrm{G}$ and $(y, x) \in \mathrm{G}'$ are equivalent.

```
assert(is_function G). red. intuition.
assert(is_function G'). red. intuition.
assert (graph G = inverse_graph(graph G')). set_extens.
assert (is_pair x). app Ha. assert (J (P x) (Q x) = x). aw.
```

```
  wr H15. wri H15 H13. aw. assert (inc (P x) (source G)).
  set (aux:=graph G). rw Hc.
  ex_tac. cp (H7 _ H16). nin H17.  nin H17. cp (H6 _ _ _ H18 H13). ue.
  assert (is_pair x). assert (is_graph (inverse_graph (graph G'))).
  ap inverse_graph_is_graph. app H14. assert (J (P x) (Q x) = x). aw.
  wri H15 H13. awi H13.  wr H15.
  assert (inc (P x) (source G)). rw H1. app range_correspondence. aw.
  ex_tac. cp (H7 _ H16). nin H17.  nin H17. cp (H6 _ _ _ H13 H17). ue.
  assert(G = inverse_fun G'). uf inverse_fun. wr H1. rw H2. wr H13. sy.
  app corr_propc.
```

Bijectivity of Γ is easy.

```
  assert (bijective G). red. split. red. split. am. ir.
  cp (defined_lem H11 H15). assert (inc (W x G) (source G')). rw H2.  fprops.
  cp (H8 _ H19). nin H20. nin H20. cp (H5 _ _ _ H18 H20).
  rwi H17 H20.  cp (defined_lem H11 H16). cp (H5 _ _ _ H23 H20). rww H24.
  ap surjective_pr5. am. ir. wri H2 H15. cp (H8 _ H15). nin H16.
  nin  H16. uf related. ex_tac. rw Hc. aw.  ex_tac.
  split. am. split. assert (G' = inverse_fun G). rw H14.
  rw inverse_fun_involutive. tv. am. rw H16. app inverse_bij_is_bij1. am.
Qed.
```

9. *Let* A, B, C, D *be sets,* f *a mapping of* A *into* B, g *a mapping of* B *into* C, h *a mapping of* C *into* D. *If* $g \circ f$ *and* $h \circ g$ *are bijections, show that all of* f, g, h *are bijections.*

```
Lemma exercise3_9: forall  f g h,
  is_function f -> is_function g -> is_function h->
  source g = target f -> source h = target g ->
  bijective(compose g f) -> bijective (compose h g) ->
  (bijective f & bijective g & bijective h).
Proof. ir. assert (composable g f). red. intuition.
  assert (composable h g). red. intuition.
  assert(injective g). red in H5; nin H5.  app (inj_right_compose H7 H5).
  assert(surjective g). red in H4; nin H4.  app (surj_left_compose H6 H9).
  assert (bijective g). red. intuition. split.
  app (bij_right_compose H6 H4 H10). split. am.
  app (bij_left_compose H7 H5 H10).
Qed.
```

10. *Let* A, B, C *be sets,* f *a mapping of* A *into* B, g *a mapping of* B *into* C, h *a mapping of* C *into* A. *Show that if two of the three mappings* $h \circ g \circ f$, $g \circ f \circ h$, $f \circ h \circ g$ *are surjections and the third is an injection, then* f, g, h *are all bijections.*

The French version claims that the same conclusion holds if two of the three mappings are injections and the third is a surjection. We assume here $h \circ g \circ f$ injective, $g \circ f \circ h$ surjective and $f \circ h \circ g$ injective or surjective. Other cases are equivalent, by renaming variables.

```
Lemma exercise3_10: forall  f g h,
  is_function f -> is_function g -> is_function h->
  source g = target f -> source h = target g ->  source f = target h ->
  injective (compose h (compose g f)) ->
  surjective (compose g (compose f h)) ->
  (injective (compose f (compose h  g))
```

```
   \/ surjective (compose f (compose h  g))) ->
  (bijective f & bijective g & bijective h).
Proof. ir.
  assert (composable f h). red. ee; am.
  assert (composable h g). red. ee; am.
  assert (composable g f). red; ee; am.
  wri compose_assoc H5; try am.
  assert (is_function (compose h g)).  fct_tac.
  assert (composable (compose h g) f). red. au.
  cp (inj_right_compose H12 H5).
  assert (is_function (compose f h)). fct_tac.
  assert (composable g (compose f h)). red. au.
  cp (surj_left_compose H15 H6).
```

In both cases we know that f is injective and g surjective. If $f \circ h \circ g$ is injective, we deduce g injective; but surjectivity of g says $f \circ h$ injective; hence f is surjective. Injectivity of g in the second relation says $f \circ h$ surjective. Thus f, g and $f \circ h$ are injective and surjective; the result follows.

```
  nin H7.
  wri compose_assoc H7; try am.
  assert (composable (compose f h) g). red. au.
  cp (inj_right_compose H17 H7).
  cp (surj_left_compose2 H15 H6 H18).
  cp (surj_left_compose H8 H19).
  cp (inj_left_compose2 H17 H7 H16).
  assert (bijective (compose f h)). red. au.
  assert (bijective f). red. ee;am. au.
  ee. am. split; am. app (bij_right_compose H8 H22 H23).
```

The second case is similar.

```
  assert (composable f (compose h g)). red. au.
  cp (surj_left_compose H17 H7).
  cp (surj_left_compose2 H17 H7 H13).
  cp (inj_left_compose2 H12 H5 H18).
  cp (inj_right_compose H9 H20). assert (bijective g). red; au.
  ee. red; ee; am. am.
  assert (bijective (compose h g)). red. ee;am.
  app (bij_left_compose H9 H23 H22).
Qed.
```

11. **Find the error in the following argument: let* **N** *denote the set of all natural numbers and let* A *denote the set of all integers $n > 2$ for which there exists three strictly positive integers x, y, z such that $x^n + y^n = z^n$. Then the set* A *is not empty (in other words, "Fermat's last theorem" is false). For let* $\mathrm{B} = \{\mathrm{A}\}$ *and* $\mathrm{C} = \{\mathbf{N}\}$*;* B *and* C *are sets consisting of a single element, hence there is a bijection f of* B *onto* C*. We have $f(\mathrm{A}) = \mathbf{N}$; if* A *were empty we would have $\mathbf{N} = f(\emptyset) = \emptyset$ which is absurd.**

We have $f\langle\emptyset\rangle = \emptyset$ and $f(\emptyset) = \mathbf{N}$. Writing the first relation as $f(\emptyset) = \emptyset$ creates an ambiguity, but has not as consequence that $\emptyset$ is equal to **N**.

8.4 Section 4

1. *Let* G *be a graph. Show that the following three propositions are equivalent: (a)* G *is a functional graph, (b) if* X, Y *are any two sets, then* $G^{-1}(X \cap Y) = G^{-1}(X) \cap G^{-1}(Y)$. *(c) The relation* $X \cap Y = \emptyset$ *implies* $G^{-1}(X) \cap G^{-1}(Y) = \emptyset$.

```
Lemma exercise4_1a: forall g,  is_graph g  ->
  functional_graph g = (forall x y, inv_image_by_graph g (intersection2 x y)=
    intersection2 (inv_image_by_graph g x) (inv_image_by_graph g y)).
Proof. ir. assert (is_graph (inverse_graph g)).  fprops.
  uf inv_image_by_graph. app iff_eq. ir. set_extens. awi H2.
  nin H2. nin H2. app intersection2_inc; aw; ex_tac; inter2tac.
  nin (intersection2_both H2). awi H3. awi H4.
  nin H3; nin H4. nin H3; nin H4. awi H5; awi H6.
  cp (H1 _ _ _ H5 H6). aw. exists x1. split.
  app intersection2_inc. rww H7. aw.

  ir. red. ir. cp (H1 (singleton y)(singleton y')).
  set (u:=intersection2 (singleton y) (singleton y')).
  assert (inc x (image_by_graph (inverse_graph g) u)). uf u. rw H4.
  app intersection2_inc. aw. ex_tac. aw. aw. ex_tac. aw.
  awi H5. nin H5. nin H5.
  ufi u H5. nin (intersection2_both H5). awi H7.  awi H8. ue.
Qed.

Lemma exercise4_1b: forall g,  is_graph g  ->
  functional_graph g = (forall x y, intersection2 x y = emptyset ->
    intersection2 (inv_image_by_graph g x) (inv_image_by_graph g y)=emptyset).
Proof. ir. app iff_eq. ir. rwi exercise4_1a H0. wr H0. rw H1.
  uf inv_image_by_graph. rww image_by_emptyset. am.
  ir. red. ir. assert (is_graph (inverse_graph g)). fprops.
  set (v:= intersection2 (image_by_graph (inverse_graph g) (singleton y))
         (image_by_graph (inverse_graph g) (singleton y'))).
  assert (inc x v). uf v. app intersection2_inc. aw. ex_tac.
  aw. aw. ex_tac. aw.
  nin (emptyset_dichot (intersection2 (singleton y) (singleton y'))).
  cp (H0 _ _ H5). assert (inc x emptyset). wr H6. am.
  elim (emptyset_pr H7). nin H5.
  nin (intersection2_both H5). awi H6. awi H7. ue.
Qed.
```

2. *Let* G *be a graph. Show that for each set* X *we have* $G(X) = \mathrm{pr}_2(G \cap (X \times \mathrm{pr}_2 G))$ *and* $G(X) = G(X \cap \mathrm{pr}_1 G)$.

```
Lemma exercise4_2a: forall g x,  is_graph g  ->
  image_by_graph g x = range(intersection2 g (product x (range g))).
Proof. ir. assert(is_graph (intersection2 g (product x (range g)))).
  red. ir. app H. inter2tac.
  set_extens. awi H1. nin H1. nin H1.  aw. exists x1.
  app intersection2_inc. aw. ee. fprops. am. ex_tac.
  awi H1. nin H1. nin (intersection2_both H1).
  aw. ex_tac. awi H3. ee. am. am. am.
Qed.
Lemma exercise4_2b: forall g x,  is_graph g  ->
```

```
  image_by_graph g x = image_by_graph g (intersection2 x (domain g)).
Proof. ir. set_extens. awi H0. nin H0. nin H0. aw. exists x1. split.
  app intersection2_inc. ex_tac. am.
  awi H0. nin H0. nin H0. aw. ex_tac. inter2tac.
Qed.
```

3. *Let* X, Y, Y′, Z *be four sets. Show that* $(Y' \times Z) \circ (X \times Y) = \emptyset$ *if* $Y \cap Y' = \emptyset$ *and that* $(Y' \times Z) \circ (X \times Y) = X \times Z$ *if* $Y \cap Y' \neq \emptyset$.

```
Lemma exercise4_3a: forall  x y y' z, intersection2 y y' = emptyset ->
  compose_graph(product y' z)(product x y) = emptyset.
Proof. ir.  app is_emptyset. ir. red. ir. awi H0.
  nin H0. nin H1. nin H1. awi H1. awi H2. ee.
  elim (emptyset_pr (x:=x0)).  wr H. app intersection2_inc.
Qed.
Lemma exercise4_3b: forall  x y y' z, nonempty(intersection2 y y') ->
  compose_graph(product y' z)(product x y) = product x z.
Proof. ir.  nin H. set_extens. awi H0. nin H0. nin H1. nin H1. awi H1; awi H2.
  aw. ee; am.
  awi H0. ee. nin (intersection2_both H). aw. split. am. exists y0. fprops.
Qed.
```

4. *Let* $(G_\iota)_{\iota \in I}$ *be a family of graphs. Show that for every set* X *we have* $(\bigcup_{\iota \in I} G_\iota)\langle X \rangle = \bigcup_{\iota \in I} G_\iota \langle X \rangle$, *and that for every object* x, $(\bigcap_{\iota \in I} G_\iota)\langle \{x\} \rangle = \bigcap_{\iota \in I} G_\iota \langle \{x\} \rangle$. *Give an example of two graphs* G, H *and a set* X *such that* $(G \cap H)\langle X \rangle \neq G\langle X \rangle \cap H\langle X \rangle$.

We have to show that $(\exists y \in X)(\exists i)(x, y) \in G_i$ is the same as $(\exists i)(\exists y \in X)(x, y) \in G_i$.

```
Lemma exercise4_4a:  forall g x,
  image_by_graph(unionb g) x =
  unionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Proof. ir.
  set_extens. awi H. nin H. nin H. rwi unionb_rw H0.
  nin H0. nin H0. apply unionb_inc with x2. bw. bw. aw.  ex_tac.
  rwi unionb_rw H. nin H. nin H. bwi H.  bwi H0. awi H0. nin H0.
  nin H0. aw. ex_tac. apply unionb_inc with x1. am. am. am.
Qed.
```

We have to show that $(\exists y \in X)(\forall i)(x, y) \in G_i$ is the same as $(\forall i)(\exists y \in X)(x, y) \in G_i$. We cannot exchange quantifiers. However, if X is a singleton $\{u\}$, $y \in X$ is equivalent to $y = u$, and this commutes. We need an auxiliary result: $G_\iota\langle \{x\} \rangle$ is a nonempty family.

```
Lemma exercise4_4b:  forall g x,
  nonempty g -> is_singleton x ->
  image_by_graph(intersectionb g) x =
  intersectionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Proof. ir.
  red in H0. nin H0. rw H0.
  assert (Hb:nonempty
    (L (domain g) (fun i => image_by_graph (V i g) (singleton x0)))).
  cp (nonempty_domain H).  nin H1.
  set (ff:=  (fun i=> image_by_graph (V i g) (singleton x0))).
  exists (J y (ff y)). uf L. aw. exists y. au.
  set_extens. awi H1. nin H1. nin H1. cp (singleton_eq H1). rwi H3 H2.
```

```
  rwi intersectionb_rw H2.
  ap intersectionb_inc. am. bw. ir. bw. aw. ex_tac. rw H3. app H2. am.
  aw. ex_tac. apply intersectionb_inc. am. ir.
  rwi intersectionb_rw H1. bwi H1. cp (H1 _ H2). bwi H3.
  awi H3. nin H3. nin H3. rwi (singleton_eq H3) H4. am. am. am.
Qed.
```

Let us turn now to the example. We want to find X, G and H such that $p(X) \neq q(X)$. We have $p(X) = p(X')$ and $q(X) = q(X')$ where X' is the intersection of X and the domain of G or H. We know $p(X) = q(X)$ if X is a singleton. Thus X, G and H must have at least two elements. We give here the minimal solution: X has two elements, G is the identity in X, and H permutes the elements.

```
Lemma exercise4_4c:
  let x:=TPa in let y:= TPb in
    let G:= doubleton(J x x)(J y y) in let H:=  doubleton(J x y)(J y x)
      in let z:= doubleton x y in
        image_by_graph (intersection2 G H) z <>
        intersection2 (image_by_graph G z)(image_by_graph H z).
Proof. ir. assert (x <> y). app two_points_distinct.
  assert (Ha:is_graph G). red. ir. nin (doubleton_or H1); rw H2; fprops.
  assert (Hb:is_graph H). red. ir. nin (doubleton_or H1); rw H2; fprops.
  assert (Hc:image_by_graph G z = z). set_extens. awi H1. nin H1. nin H1.
  nin (doubleton_or H2);  cp (pr2_injective H3);rw H4; uf z; fprops.
  nin (doubleton_or H1); aw; ex_tac; rww H2; uf G; fprops.
  assert (Hd:image_by_graph H z = z). set_extens. awi H1. nin H1. nin H1.
  nin (doubleton_or H2);  cp (pr2_injective H3); rw H4;  uf z; fprops.
  aw. uf H. nin (doubleton_or H1);  [ exists y | exists x] ;
  rw H2; split; uf z;fprops;  uf H; fprops.
  assert (He: intersection2 G H = emptyset). app is_emptyset. red. ir.
  nin (intersection2_both H1). elim H0.
  nin (doubleton_or H2); nin (doubleton_or H3); rwi H4 H5;
    try rw (pr2_injective H5); try rw (pr1_injective H5); au.

  rw Hc. rw Hd. rw He. rw intersection2idem.
  assert (inc x z). uf z. fprops. red. ir. wri H2 H1. awi H1. nin H1. nin H1.
  red in H3. elim (emptyset_pr H3).
Qed.
```

5. *Let* $(G_\iota)_{\iota\in I}$ *be a family of graphs and let* H *be a graph. Show that*

$$(\bigcup_{\iota\in I} G_\iota)\circ H = \bigcup_{\iota\in I}(G_\iota\circ H) \qquad \textit{and} \qquad H\circ(\bigcup_{\iota\in I} G_\iota) = \bigcup_{\iota\in I}(H\circ G_\iota).$$

```
Lemma exercise4_5:  forall g h,
  (compose_graph (unionb g) h =
     unionb (L(domain g) (fun i=> compose_graph(V i g) h))
     & compose_graph h (unionb g) =
     unionb (L(domain g) (fun i=> compose_graph h (V i g)))).
Proof.  ir. split.
  set_extens. awi H0. nin H0. nin H1. nin H1.
  rwi unionb_rw H2. nin H2. nin H2. apply unionb_inc with x1. bw. bw. aw.
  ee. am. ex_tac.
  rwi unionb_rw H0. nin H0. nin H0. bwi H0. bwi H1. awi H1. nin H1. nin H2.
```

```
  nin H2. aw. split. am. ex_tac.
  apply unionb_inc with x0;tv. am.
```

Second part. The proof is almost identical.

```
  set_extens. awi H0. nin H0. nin H1. nin H1.
  rwi unionb_rw H1. nin H1. nin H1. apply unionb_inc with x1. bw. bw. aw.
  ee. am. ex_tac.
  rwi unionb_rw H0. nin H0. nin H0.  bwi H0. bwi H1. awi H1. nin H1. nin H2.
  nin H2. aw. split. am. ex_tac. apply unionb_inc with x0; am. am.
Qed.
```

6. *A graph* G *is functional if and only if for each pair of graphs* H, H′ *we have*

$$(H \cap H') \circ G = (H \circ G) \cap (H' \circ G).$$

Note that $(H \cap H') \circ G \subset (H \circ G) \cap (H' \circ G)$ is true for any graphs.

```
Lemma exercise4_6:  forall G, is_graph G ->
  fgraph G = (forall H H', is_graph H -> is_graph H' ->
    compose_graph (intersection2  H H') G =
    intersection2 (compose_graph H G) (compose_graph H' G)).
Proof. ir.  app iff_eq. ir.
  set_extens. awi H4. nin H4. nin H5. nin H5.
  app intersection2_inc; aw; split; tv; ex_tac; inter2tac.
  nin (intersection2_both H4).
  awi H5; awi H6.  nin H5; nin H6. nin H7;nin H8. nin H7; nin H8.
  nin H0. assert (P (J (P x) x0) = P (J (P x) x1)). aw.
  cp (H11 _ _ H7 H8 H12). cp (pr2_injective H13).
  aw. split. am. ex_tac. app intersection2_inc. ue.
```

Converse. If $(x, y) \in G$ and $(x, y') \in G$ we consider the mappings $y \mapsto x$ and $y' \mapsto x$. Then (x, x) is in $H \circ G$ and $H' \circ G$. Thus $H \cap H'$ is nonempty.

```
  ir. red. split. am. ir.
  set (h:= singleton(J (Q x) (P x))).
  set (h':= singleton(J (Q y) (P y))).
  assert (is_graph h). uf h. red. ir. inter2tac.
  assert (is_graph h'). uf h'. red. ir. inter2tac.
  cp (H _ H1). wri (pair_recov H6) H1.
  cp (H _ H2). wri (pair_recov H7) H2.
  assert (inc (J (P x)(P x)) (compose_graph h G)). aw. split. fprops.
  ex_tac. uf h.  fprops.
  assert (inc (J (P y)(P y)) (compose_graph h' G)). aw. split. fprops.
  ex_tac. uf h'. fprops.
  assert (inc (J (P x)(P x)) (compose_graph (intersection2 h h') G)).
  rww H0. app intersection2_inc. rww H3. awi H10. nin H10. nin H11. nin H11.
  nin (intersection2_both H12). ufi h H13. cp (singleton_eq H13).
  ufi h H14. cp (singleton_eq H14).
  app pair_extensionality.  rwi H15 H16. inter2tac.
Qed.
```

7. *Let* G, H, K *be three graphs. Prove the relation* $(H \circ G) \cap K \subset (H \cap (K \circ G^{-1})) \circ (G \cap (H^{-1} \circ K))$.

```
Lemma exercise4_7:  forall G H K,
  sub(intersection2 (compose_graph H G) K)
     (compose_graph(intersection2 H (compose_graph K (inverse_graph G)))
       (intersection2 G (compose_graph (inverse_graph H) K))).
Proof. ir. red. ir. nin (intersection2_both H0).
  awi H1. nin H1. nin H3. nin H3. assert (J (P x)(Q x)=x). aw.
  wri H5 H2. aw. split. am. exists x0.
  split; inter2tac; aw; split;fprops; ex_tac; aw.
Qed.

Proof. ir. red. ir. nin (intersection2_both H3). awi H4. nin H4. nin H6.
  nin H6. assert (J (P x)(Q x)=x). aw.
  wri H8 H5. aw. split. am. exists x0.
  split;  app intersection2_inc; aw; split;fprops; ex_tac; aw; fprops.
Qed.
```

8. *Let* $\mathfrak{R} = (X_\iota)_{\iota \in I}$ *and* $\mathfrak{S} = (Y_\kappa)_{\kappa \in K}$ *be two coverings of a set* E. *(a) Show that if* $\mathfrak{R}$ *and* $\mathfrak{S}$ *are partitions of* E *and if* $\mathfrak{R}$ *is finer than* $\mathfrak{S}$, *then for every* $\kappa \in K$ *there exists* $\iota \in I$ *such that* $X_\iota \subset Y_\kappa$. *(b) Give an example of two coverings* $\mathfrak{R}$ *and* $\mathfrak{S}$ *such that* $\mathfrak{R}$ *is finer than* $\mathfrak{S}$ *but such that the property stated in (a) is not satisfied. (c) Give an example of two partitions* $\mathfrak{R}$ *and* $\mathfrak{S}$ *such that for every* $\kappa \in K$ *there exists* $\iota \in I$ *such that* $X_\iota \subset Y_\kappa$, *but such that* $\mathfrak{R}$ *is not a refinement of* $\mathfrak{S}$.

The French version does not assume that $\mathfrak{R}$ is a partition. We must however assume $Y_\kappa \neq \emptyset$.

```
Lemma exercise4_8a:  forall dr r ds s x,
  covering_f dr r x -> covering_f ds s x ->
  partition_fam (L ds s) x ->  coarser_covering ds s dr r ->
  (forall k, inc k ds -> nonempty (s k)) ->
  forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k).
Proof. ir. red in H1. ee. cp (H3 _ H4). nin H7. assert (inc y x). wr H6.
  apply unionb_inc with k. bw. bw. red in H. assert (inc y (unionf dr r)).
  app H. nin (unionf_exists H9). nin H10. ex_tac.
  nin (H2 _ H10). nin H12. assert (inc y (s x1)). app H13.
  red in H5. bwi H5. nin (H5 _ _ H4 H12). ue.
  red in H15. elim (emptyset_pr (x:=y)).  wr H15. bw. app intersection2_inc.
Qed.
```

We consider a covering R, and take for S the union of R and another set. Then R is finer than S.

```
Lemma exercise4_8b: let a:= TPa in let b:= TPb in
  let x:= doubleton a b in let dr:= singleton a in let r:= fun _ => x
    in let ds:= x in let s:= variant a x (singleton a) in
      (covering_f dr r x & covering_f ds s x &
        coarser_covering ds s dr r &
        (forall k, inc k ds -> nonempty (s k)) &
        ~ (forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k))).
Proof. ir. uf ds; uf dr.
  assert (Ha: b<> a). uf a; uf b. app two_points_distinctb.
  split. red. red. ir. apply unionf_inc with a. fprops.
  uf r. am. split. red. red. ir. uf x. apply unionf_inc with a; fprops.
```

```
  uf s. rww variant_if_rw.
  split. red. ir.  exists a. split. uf x. fprops.
  uf s. rww variant_if_rw. uf r. fprops.
  split. ir. ufi x H. uf s. nin (doubleton_or H); rw H0.
  rww variant_if_rw. uf x. app nonempty_doubleton.
  rww variant_if_not_rw. app nonempty_singleton.
  red. ir. assert (inc b x). uf x. fprops. nin (H _ H0). nin H1.
  ufi r H2. ufi s H2. rwii variant_if_not_rw H2. cp (H2 _ H0). awi H3.
  contradiction.
Qed.
```

Second counter example. The mapping $\kappa \mapsto \iota$ is injective. If I and K have the same number of elements, both partitions are equivalent. If K has a single element, then R if finer than S. Thus we need S_1 and S_2, $R_1 \subset S_1$, $R_2 \subset S_2$ and R_3 that is neither in S_1 nor in S_2, thus has an element in S_1 and another one in S_2. Thus E has at least four elements; we could use $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$ and $\{\{\{\emptyset\}\}\}$, but it is a bit longish to prove that all elements are distinct. We consider here two sets of 3 and 4 elements, and a tatic that solves inequalities.

```
Inductive four_points : Set :=  | fpa  | fpb | fpc |  fpd.
Inductive three_points : Set :=  | tpa  | tpb | tpc.

Ltac disc:=
  match goal with
    |- Ro ?x <> Ro ?y =>
      red; ir;cut(x = y); [ intros hyp; discriminate hyp | app  R_inj]
    | h:Ro ?x = Ro ?y |- False =>
     cut(x = y); [ intros hyp; discriminate hyp | app  R_inj]
end.

Lemma exercise4_8c:
  let x:= four_points in let dr:= three_points in
    let r:= fun i=>  Yo (i = (Ro tpa)) (singleton (Ro fpa))
      (Yo (i = (Ro tpc)) (singleton (Ro fpb)) (doubleton (Ro fpc) (Ro fpd)))
      in let ds:= doubleton(Ro fpa) (Ro fpb)
        in let s:= variant (Ro fpa) (doubleton (Ro fpa) (Ro fpc))
          (doubleton (Ro fpb) (Ro fpd))
          in (partition_fam (L ds s) x &
            partition_fam (L dr r) x&
           (forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k))&
           ~( coarser_covering ds s dr r )).
```

The first step is to prove that S is a partition. It has two elements $S_a = \{a, c\}$ and $S_b = \{b, d\}$. The key relation is $S_a \cap S b = \emptyset$.

```
Proof. ir.
  assert (Hw: Ro fpb <> Ro fpa). disc.
  split. red. split. gprops. split.
  assert (disjoint (V (Ro fpa) (L ds s)) (V (Ro fpb) (L ds s))).
  uf ds. bw. uf s. rww variant_if_rw. rw variant_if_not_rw.
  app disjoint_pr. ir. nin (doubleton_or H);  nin (doubleton_or H0);
  rwi H1 H2; disc. exact Hw.
  fprops. fprops.
  red. bw. ir. nin (doubleton_or H0); nin (doubleton_or H1); rw H2; rw H3;
  solve [ left; tv | right; am | right; app disjoint_symmetric].
```

We show that S is a covering. For each x, there is a v such that $x \in S_v$. It is respectively a, b, a and b.

```
uf ds. uf s. set_extens. nin (unionb_exists H); nin H0; bwi H0; bwi H1;
  nin (doubleton_or H0); rwi H2 H1.
rwii variant_if_rw H1. nin (doubleton_or H1);  rw H3; app R_inc.
rwii variant_if_not_rw H1. nin (doubleton_or H1);  rw H3; app R_inc. am. am.
nin H. wr H. elim x1 ;
 [ set (v:= fpa) | set (v:= fpb) | set (v:= fpa) |  set (v:= fpb) ];
 apply unionb_inc with (Ro v); bw; fprops; bw;
 solve [  rww variant_if_rw;  fprops; fprops |
          rww variant_if_not_rw; fprops; fprops ].
```

We prove now that R is a partition. Since R has three elements it is a bit longer (we must show that 6 pairs of sets are disjoint). We have $R_a = \{a\}$ and $R_b = \{c, d\}$, $R_c = \{b\}$.

```
assert (Ha:disjoint (r (Ro tpa)) (r (Ro tpb))).
uf r. rww Y_if_rw. rww Y_if_not_rw. rww Y_if_not_rw.
app disjoint_pr. ir. awi H.  rwi H H0.
nin (doubleton_or H0); disc. disc. disc.
assert (Hb:disjoint (r (Ro tpa)) (r (Ro tpc))).
uf r. rww Y_if_rw.  rww Y_if_not_rw. rww Y_if_rw.
app disjoint_pr. ir. awi H. awi H0. rwi H0 H.  disc. disc.
assert (Hc:disjoint (r (Ro tpc))  (r (Ro tpb))).
uf r. rww Y_if_not_rw. rww Y_if_rw.  rww Y_if_not_rw. rww Y_if_not_rw.
app disjoint_pr. ir. awi H.  rwi H H0.
nin (doubleton_or H0); disc. disc. disc. disc.
split. red. split. gprops. split.
red. ir. bwi H. bwi H0. bw. ufi dr H.
ufi dr H0.  nin H; nin H0. wr H; wr H0. elim x0; elim x1;
 solve [ left; tv | right; am | right; app disjoint_symmetric].
```

We now show that R is a covering. For each x, there is a v such that $x \in R_v$. It is respectively a, c, b and b.

```
set_extens. nin (unionb_exists H). nin H0. bwi H0. bwi H1. ufi r H1.
ufi dr H0. nin H0. wri H0 H1. induction x2. rwii Y_if_rw H1.
rw (singleton_eq H1). app R_inc. rwi Y_if_not_rw H1. rwi Y_if_not_rw H1.
nin (doubleton_or H1);rw H2; app R_inc. disc. disc.
rwi Y_if_not_rw H1. rwi Y_if_rw H1.  rw (singleton_eq H1).
app R_inc. tv.  disc. am.
uf r. nin H. wr H.
induction x1 ; [set (v:= tpa) | set (v := tpc) | set (v := tpb)
  | set (v:= tpb) ] ; apply unionb_inc with (Ro v); bw; try app R_inc.
rww Y_if_rw. fprops.
rw Y_if_not_rw. rw Y_if_rw. fprops. tv. disc.
rw Y_if_not_rw. rw Y_if_not_rw. fprops. disc.  disc.
rw Y_if_not_rw. rw Y_if_not_rw. fprops. disc.  disc.
```

We show that for all $\kappa \in K$ there exists $\iota \in I$ such that $X_\iota \subset Y_\kappa$. This is $R_a \subset S_a$ and $R_c \subset S_b$.

```
split. uf s. uf r.  ir. ufi ds H. nin (doubleton_or H). rw H0.
rww variant_if_rw. exists (Ro tpa). split. app R_inc.
rww Y_if_rw.  red. ir. inter2tac.
rw H0. rw variant_if_not_rw. exists (Ro tpc). split. app R_inc.
rw Y_if_not_rw. rww Y_if_rw. red. ir. inter2tac. disc. disc.
```

Now, we show that R_b is not a subset of any S_ι.

```
  red. ir. red in H. induction (H _ (R_inc tpb)).  nin H0. ufi r H1.
  rwi Y_if_not_rw H1. rwi Y_if_not_rw H1.
  assert (inc (Ro fpc) (s x0)). app H1. fprops.
  assert (inc (Ro fpd) (s x0)). app H1. fprops.
  ufi ds H0. nin (doubleton_or H0). rwi H4 H3. ufi s H3.
  rwi variant_if_rw H3. nin (doubleton_or H3);  disc. tv.
  rwi H4 H2. ufi s H2. rwi variant_if_not_rw H2.
  nin (doubleton_or H2); disc. am. disc. disc.
Qed.
```

8.5 Section 5

* *Montrer que si* X, Y *sont deux ensembles tels que* $\mathfrak{P}(X) \subset \mathfrak{P}(Y)$, *on a* $X \subset Y$.
This exercise appears only in the French version.

```
Lemma exercise5_f1: forall x y, sub(powerset x) (powerset y) -> sub x y.
Proof. ir. red. ir. assert (sub (singleton x0) y). app powerset_sub. app H.
  app powerset_inc. red. ir. inter2tac. red in H1.
  app (H1 x0). fprops.
Qed.
```

* *Soient* E *un ensemble* f *une application de* $\mathfrak{P}(E)$ *dans lui-même telle que la relation* $X \subset Y$ *entraîne* $f(X) \subset f(Y)$. *Soit* V *l'intersection des ensembles* $Z \subset E$ *tels que* $f(Z) \subset Z$ *et soit* W *la réunion des ensembles* $Z \subset E$ *tels que* $Z \subset f(Z)$. *Montrer que* $f(V) = V$ *et* $W = f(W)$ *et que pour tout ensemble* $Z \subset E$ *tel que* $f(Z) = Z$ *on a* $V \subset Z \subset W$.

This exercise appears only in the French version. It says: let E be set and f a mapping from $\mathfrak{P}(E)$ into itself such that $X \subset Y$ implies $f(X) \subset f(Y)$. Let V be the intersection of the sets $Z \subset E$ for which $f(Z) \subset Z$ and let W be the union of the sets $Z \subset E$ such that $Z \subset f(Z)$. Show that $f(V) = V$ and $W = f(W)$ and that for every set $Z \subset E$ such that $f(Z) = Z$ one has $V \subset Z \subset W$.

```
Lemma exercise5_f2: forall f x v w,
  is_function f -> source f = (powerset x) -> target f = powerset x ->
  (forall a b, inc a (powerset x) -> inc b (powerset x) -> sub a b
    -> sub (W a f) (W b f)) ->
  v = intersection(Zo (powerset x) (fun z=> sub (W z f) z)) ->
  w = union(Zo (powerset x) (fun z=> sub z (W z f))) ->
  (W v f = v & W w f = w & (forall z, sub z x -> W z f = z ->
    (sub v z & sub z w))).
Proof. ir.
  set (q:= (Zo (powerset x) (fun z => sub (W z f) z))).
  assert (nonempty q). uf q. exists x. Ztac. app inc_x_powerset_x.
  app powerset_sub. wr H1. app inc_W_target. rw H0.  app inc_x_powerset_x.
  set (p:= (Zo (powerset x) (fun z => sub z (W z f)))).
  assert (Ha:forall z, sub z x -> W z f = z -> sub v z). ir.
  assert (inc z q). uf q. Ztac. app powerset_inc. rw H7. fprops. rw H3.
  fold q. app intersection_sub.
  assert (Hb:forall z, sub z x -> W z f = z -> sub z w). ir.
  assert (inc z p). uf p. Ztac. app powerset_inc. rw H7. fprops. rw H4.
  fold p. app union_sub.
  assert (Hc:forall z, inc z q -> inc (W z f) q). ir. ufi q H6. Ztac. clear H6.
```

```
  assert (inc (W z f) (powerset x)).  wr H1. wri H0 H7.  fprops.  uf q. Ztac.
  assert (Hd:forall z, inc z p -> inc (W z f) p). ir. ufi p H6. Ztac. clear H6.
  assert (inc (W z f) (powerset x)).  wr H1. wri H0 H7.  fprops. uf p. Ztac.
  assert (He:inc v (powerset x)). app powerset_inc. rw H3. red. ir.
  nin H5. fold q in H6.  cp (intersection_forall H6 H5). unfold q in H5.
  Ztac. cp (powerset_sub H8). app H10.
  assert (Hf:inc w (powerset x)). app powerset_inc. rw H4. red. ir.
  cp (union_exists H6). nin H7.  nin H7. Ztac. cp (powerset_sub H9). app H11.
  assert (Hg:sub (W v f) v). red. ir. rw H3. app intersection_inc. ir.
  fold q in H7.  assert (sub v y). rw H3. app intersection_sub. ufi q H7. Ztac.
  cp (H2 _ _ He H9 H8). app H10.  app H11.
  assert (Hh:sub w (W w f)). red. ir. rwi H4 H6. cp (union_exists H6). nin H7.
  nin H7. assert (sub x1 w). rw H4. app union_sub. Ztac.
  cp (H2 _ _ H10 Hf H9). app H12. app H11.
  split. apply extensionality. am. assert (inc v q). uf q. Ztac. cp (Hc _ H6).
  set (k:= W v f). rw H3. app intersection_sub.
  split. apply extensionality.  assert (inc w p). uf p. Ztac. cp (Hd _ H6).
  set (k:= W w f). rw H4. app union_sub. am. ir. split. app Ha. app Hb.
Qed.
```

1. *Let* $(X_\iota)_{\iota\in I}$ *be a family of sets. Show that if* $(Y_\iota)_{\iota\in I}$ *is a family of sets such that* $Y_\iota \subset X_\iota$ *for each* $\iota \in I$ *then* $\prod_{\iota\in I} Y_i = \bigcap_{\iota\in I} \mathrm{pr}_\iota^{-1}(Y_\iota)$.

```
Lemma exercise5_1: forall id x y,
  (forall i, inc i id -> sub (y i) (x i)) -> nonempty id ->
  productf id y =
  intersectionf id (fun i=> inv_image_by_fun (pr_i (L id x) i) (y i)).
Proof. ir. set_extens. app intersectionf_inc. ir. uf inv_image_by_fun.
  assert (Ha: fgraph (L id x)). gprops.
  assert (Hb:is_function (pr_i (L id x) j)). app function_pri. bw.
  rwi productf_pr H1. ee. aw. exists (V j x0). split.  app H4. ue.
  assert (inc j (domain  (L id x))). bw.
  assert(inc x0 (productb (L id x))).  rw productb_pr. split. am.
  split. bw. ir. rwi H3 H6. bw. app (H _ H6). app H4. ue. am.
  wr (W_pri Ha H5 H6). graph_tac.
  assert (inc (rep id) id). app nonempty_rep.
  cp (intersectionf_forall H1 H2). simpl in H3. ufi inv_image_by_fun H3.
  assert (Ha: fgraph (L id x)).  gprops.
  assert (Hb:is_function (pr_i (L id x) (rep id))). app function_pri. bw.
  awi H3. nin H3. nin  H3. red in H4. cp (inc_pr1graph_source Hb H4).
  simpl in H5. rwii productb_pr H5. nin H5; nin H6.
  rw productf_pr. bwi H6.  split. am. split. am. ir.
  rwi H6 H8. cp (intersectionf_forall H1 H8). simpl in H9.
  ufi inv_image_by_fun H9.
  assert (Hc:is_function (pr_i (L id x) i)). app function_pri. bw.
  awi H9. nin H9. nin  H9. red in H10. cp (W_pr Hc H10).
  rwi W_pri H11. ue. am. bw. rw productb_pr. intuition. bw. am.
Qed.
```

2. *Let* A, B *be two sets. For each subset* G *of* $A \times B$ *let* $\tilde{G}$ *be the mapping* $x \mapsto G(\{x\})$ *of* A *into* $\mathfrak{P}(B)$. *Show that the mapping* $G \mapsto \tilde{G}$ *is a bijection from* $\mathfrak{P}(A \times B)$ *onto* $(\mathfrak{P}(B))^A$.

Note that $\tilde{G}$ is in $\mathscr{F}(A;\mathfrak{P}(B))$. The French edition says: let $\tilde{G}$ be the graph of the mapping etc, so that $\tilde{G}$ is in $(\mathfrak{P}(B))^A$.

```
Lemma exercise5_2: forall a b,
  bijective (BL(fun g => L a (fun x => image_by_graph g (singleton x)))
    (powerset (product a b)) (set_of_gfunctions a (powerset b))).
Proof. ir.
  set(tilde:=(BL (fun g => L a (fun x => image_by_graph g (singleton x)))
    (powerset (product a b)) (set_of_gfunctions a (powerset b)))).
```

We first prove that the mapping $G \mapsto \tilde{G}$ is a function.

```
  assert (transf_axioms
    (fun g => L a (fun x => image_by_graph g (singleton x)))
    (powerset (product a b)) (set_of_gfunctions a (powerset b))).
  red. ir. cp (powerset_sub H).
  set (faux:=BL(fun x=> image_by_graph c (singleton x)) a (powerset b)).
  assert(is_function faux).  uf faux. app af_function. red. ir.
  ap powerset_inc. red. ir. awi H2. nin H2.  nin H2. red in H3. cp (H0 _ H3).
  rwi inc_product H4. awi H4. intuition.
  change (inc (graph faux) (set_of_gfunctions (source faux) (target faux))).
  app inc_set_of_gfunctions.
```

We prove that the mapping is injective.

```
  uf tilde. app bijective_af_function. ir.
  set (fx:= L a (fun x0 => image_by_graph u (singleton x0))).
  set (fy:= L a (fun x0 => image_by_graph v (singleton x0))).
  cp (powerset_sub H0).  cp (powerset_sub H1).
  set_extens. cp (H3 _ H5). awi H6; ee.
  assert (inc (Q x) (V (P x) fy)). uf fy. wr H2. bw. aw.
  ex_tac. aw.  ufi fy H9. bwi H9. awi H9. nin H9.
  nin H9. red in H10.  rwi (singleton_eq H9) H10. aw. awi H10. am. am. am.
  cp (H4 _ H5). awi H6. ee.
  assert (inc (Q x) (V (P x) fx)). uf fx. rw H2. bw. aw.
  ex_tac. aw. ufi fx H9. bwi H9. awi H9. nin H9.
  nin H9. red in H10.  rwi (singleton_eq H9) H10. awi H10. am. am. am.
```

We prove that the mapping is surjective.

```
  ir. nin (set_of_gfunctions_inc H0). nin H1. ee.
  set (g:=Zo (product a b) (fun z => inc (Q z) (V (P z) y))).
  assert(inc g (powerset (product a b))).  app powerset_inc. red.
  uf g. ir. Ztac. am.
  assert (Ha:is_graph g). red. uf g. ir. Ztac. awi H7. nin H7. am.
  ex_tac. uf tilde. aw.  app function_extensionality.
  gprops. wr H4. fprops. bw. wr H4. sy. aw.
  ir.  assert (inc x0 a). bwi H6. am.  bw.
  sy. set_extens. change (inc x1 (im_singleton g x0)). rw im_singleton_pr. uf g. Ztac.
  app product_pair_inc. assert (sub (V x0 y) b). wr H4.
  change (sub (W x0 x) b).  wr powerset_inc_rw. wr H3. wri H2 H7. fprops.
  app H9. aw. awi H8. nin H8. nin H8. rwi (singleton_eq H8) H9.
  ufi g H9. Ztac. awi H11. am.
Qed.
```

3. ** Let $(X_i)_{1\le i\le n}$ be a finite family of sets. For each subset H of the index set $[1,n]$ let $P_H = \bigcup_{i\in H} X_i$ and $Q_H = \bigcap_{i\in H} X_i$. Let $\mathfrak{F}_k$ be the set of subsets of $[1,n]$ which have k elements. Show*

that

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \supset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \le (n+1)/2$$

and that

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \subset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \ge (n+1)/2. \quad *$$

Bourbaki defines integers and finite sets only later. We can define finite sets by induction. We could try to say: let I be the index, T be a subset of I, and consider all H equipotent to T. The condition $k \le n/2$ is equivalent to: there is an injection from T into the complementary. The condition is however $k \le (n+1)/2$, this makes the result non-obvious.

8.6 Section 6

1. *For a graph* G *to be the graph of an equivalence relation on a set* E*, it is necessary and sufficient that* $\mathrm{pr}_1 G = E$, $\mathrm{pr}_2 G = E$, $G \circ G^{-1} \circ G = G$ *and* $\Delta_E \subset G$ *(*Δ_E *being the diagonal of* E*).*

Comment. The condition $\mathrm{pr}_2 G = E$ was missing in the English version [2]. It is necessary: consider the graph with two elements (a,a) and (a,b).

```
Lemma exercise6_1: forall x g, is_graph g ->
  (is_equivalence g & substrate g = x) =
  (domain g = x & range g = x &
    compose_graph g (compose_graph (inverse_graph g) g) = g
    & sub (diagonal x) g).
Proof. ir. app iff_eq. ir. nin H0. cp (domain_is_substrate H0).
  ir.  ee. ue. wr H1. uf substrate. set_extens. inter2tac.
  nin (union2_or H3). aw.  exists x0.
  rwi H2 H4. equiv_tac. am.
  set_extens. awi H3. nin H3. nin H4. nin H4. awi H4. nin H4. nin H6. nin H6.
  awi H7. assert (inc (J x2 x1) g). equiv_tac. assert (inc (J (P x0) x1) g).
  equiv_tac. assert (inc (J (P x0) (Q x0)) g).  equiv_tac.  awi H10. am. am.
  assert (is_pair x0). app H. assert (J (P x0) (Q x0) = x0). aw.
  assert (inc (J (P x0) (P x0)) g). equiv_tac. substr_tac.
  aw. split. am. exists (P x0). split. aw. split. fprops.
  ex_tac. aw. ue.
  red. ir. rwi inc_diagonal H3.  ee.
  assert (J (P x0) (Q x0) = x0). aw. wr H6. wr H5.  wri H1 H4. equiv_tac.
```

Now the converse

```
  ir. assert (substrate g  = x). ee. uf substrate. rw H0. rw H1. ap union2idem.
  assert (forall u, inc u x -> inc (J u u) g). ir. ee. app H5.
  rw inc_diagonal. split.  fprops. aw. au.
  assert (is_symmetric g). red. split. am. ir. red in H3. ee.
  assert (inc (J y y) g). app H2. wr H4. aw. ex_tac.
  assert (inc (J x0 x0) g). app H2. wr H0. aw. ex_tac.
  red. wr H5. aw. split. fprops. ex_tac. aw.
  split. fprops. ex_tac. aw.
  intuition. red. ee; tv;split;tv. ir. red. app H2. ue.
  ir. red. wr H5. aw. split. fprops. red in H8. ex_tac. aw.
  split. fprops. red in H6. ex_tac. aw. app H2. wr H0. aw.  ex_tac.
Qed.
```

2. *If* G *is a graph such that* $G \circ G^{-1} \circ G = G$ *show that* $G^{-1} \circ G$ *and* $G \circ G^{-1}$ *are graphs of equivalences on* $\mathrm{pr}_1 G$ *and* $\mathrm{pr}_2 G$ *respectively.*

We first compute the substrate of the relations.

```
Lemma exercise6_2: forall g, is_graph g ->
  compose_graph g (compose_graph (inverse_graph g) g) = g ->
  (is_equivalence (compose_graph (inverse_graph g) g) &
    substrate (compose_graph (inverse_graph g) g) = domain g &
    is_equivalence (compose_graph g (inverse_graph g)) &
    substrate (compose_graph g (inverse_graph g)) = range g).
Proof. ir.
  assert(Ha:is_graph (inverse_graph g)). app inverse_graph_is_graph.
  assert (Hb:is_graph (compose_graph (inverse_graph g) g)).
  app composition_is_graph.
  assert (Hc:is_graph (compose_graph g (inverse_graph g))).
  app composition_is_graph.
  assert (forall x y z t, related g x y -> related g z y -> related g z t ->
    related g x t). ir. red. wr H0. aw. split. fprops. exists z.
  split. aw. split. fprops. exists y. split. am. aw. am.
  assert (Hd:substrate (compose_graph (inverse_graph g) g) = domain g).
  set_extens. rwi inc_substrate_rw H2. nin H2. nin H2. awi H2. nin H2. nin H3.
  nin H3. ex_tac.
  nin H2. awi H2.  nin H2. nin H3. nin H3. awi H4. ex_tac. am.
  awi H2. nin H2. wri H0 H2.  awi H2. nin H2. nin H3. nin H3.
  cp (inc_pr1_substrate H3). awi H5. am. am.
  assert (He:substrate (compose_graph g (inverse_graph g)) = range g).
  set_extens. rwi inc_substrate_rw H2. nin H2. nin H2. awi H2. nin H2. nin H3.
  nin H3. awi H3. ex_tac.
  nin H2. awi H2.  nin H2. nin H3. nin H3. ex_tac. am.
  awi H2. nin H2. wri H0 H2.  awi H2. nin H2. nin H3. nin H3. awi H3.
  nin H3. nin H5. nin H5.
  assert (inc (J x2 x) (compose_graph g (inverse_graph g))). aw.
  split. fprops. exists x1. split;am.
  cp (inc_pr2_substrate H7). awi H8. am. am.
```

We apply proposition 1. Γ is an equivalence if $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. If Γ is the composition of G and G^{-1} in any order, the relation is true. The second is a consequence of the assumption and associativity of composition.

```
  set (h:=  inverse_graph g).
  split. rw equivalence_pr.  split. wr composition_associative. uf h. ue.
  rw inverse_compose.  fold h. assert (g = inverse_graph h). uf h.
  rww inverse_graph_involutive. ue.
  split. am. split. rw equivalence_pr.  split. rw composition_associative.
  rwi composition_associative H0. fold  h in H0. rww H0.
  rw inverse_compose. fold h. assert (g = inverse_graph h). uf h.
  rww inverse_graph_involutive. wrr H2. am.
Qed.
```

3. *Let* E *be a set,* A *a subset of* E*, and* R *the equivalence relation associated with the mapping* $X \mapsto X \cap A$ *of* $\mathfrak{P}(E)$ *into* $\mathfrak{P}(E)$*. Show that there exists a bijection from* $\mathfrak{P}(A)$ *onto the quotient set* $\mathfrak{P}(E)/R$*.*

If ~ is the equivalence associated, then B and B′ are related if they have the same intersection with A. If $u \in$ A, we can consider the set of all B whose intersection with A is u as a class. This is our bijection (called canonical in the French edition).

```
Definition intersection_with x a :=
  BL(fun w=> intersection2 w a) (powerset x)(powerset x).
Definition intersection_with_canon x a :=
  BL (fun b => Zo(powerset x)(fun c=> intersection2 c a = b))
  (powerset a)(quotient (equivalence_associated (intersection_with x a))).
```

We first show that we have a function.

```
Lemma exercise6_3: forall a x,
  sub a x -> bijective (intersection_with_canon x a).
Proof. ir.
  assert(Ha: forall u, sub u a -> intersection2 u a = u).
  ir. app extensionality. app intersection2sub_first.
  red. ir. app intersection2_inc. app H0.
  assert (Hb:transf_axioms (fun w0 => intersection2 w0 a)
    (powerset x) (powerset x)). red. ir. app powerset_inc.
  apply sub_trans with c. app intersection2sub_first. app powerset_sub.
  assert (is_function (intersection_with x a)).
  uf intersection_with. app af_function.
  set(r:= equivalence_associated (intersection_with x a)).
  assert (is_equivalence r). uf r. app equivalence_graph_ea.
  assert (transf_axioms (fun b => Zo(powerset x)
    (fun c=> intersection2 c a = b)) (powerset a)(quotient r)).
  red. ir. cp (powerset_sub H2).
  set (w:= Zo (powerset x) (fun c0 => intersection2 c0 a = c)).
  assert(nonempty w). exists c. uf w. Ztac. app powerset_inc.
  apply sub_trans with a. am. am.
  assert (sub w (powerset x)). uf w. app Z_sub.
  assert(inc (rep w) (powerset x)).  app H5. app nonempty_rep.
  rw inc_quotient. red. split. am. split. uf r. rw substrate_graph_ea. am. am.
  assert(intersection2 (rep w) a = c).
  set (b:= rep w). assert (inc b w). uf b. app nonempty_rep. ufi w H7.
  Ztac. rw H9.  tv.
  set_extens. bw. uf r. rw related_ea. uf intersection_with.
  simpl. split. am. split. app H5. aw. ufi w H8. Ztac. rww H10.
  app H5. am.  bwi H8. ufi r H8. rwi related_ea H8.
  nin H8.  nin H9.
  ufi intersection_with H10. awi H10. uf w. Ztac. wr H10. am. am. am. am. am.
  am. am. am.
  uf intersection_with_canon.
```

We prove injectivity.

```
  red.  split. red. split. app af_function. simpl. ir.
  awi H5.  cp (powerset_sub H3). cp (powerset_sub H4).
  assert (inc x0 (Zo (powerset x) (fun c => intersection2 c a = y))).
  wr H5. Ztac. app powerset_inc. apply sub_trans with a. am. am. Ztac.
  sy. wr H10.  app Ha. am. am. am. am.
```

We prove now the surjectivity.

```
  app surjective_pr6. app af_function. simpl.
```

```
  ir. rwi inc_quotient H3. nin H3. clear H3. nin H4.
  rwi substrate_graph_ea H3. simpl in H3.
  assert(inc (intersection2 (rep y) a) (powerset a)). app powerset_inc.
  app intersection2sub_second.
  cp (powerset_sub H5). ex_tac. aw.
  set_extens. Ztac. rw H4. bw. rw related_ea. ee; tv.
  uf intersection_with. sy; aw. am.
  rwi H4 H7.  bwi H7.  bwi H7. ufi intersection_with H7.
  rwi related_ea H7. ee. simpl in H8. awi H9. Ztac. am. am. am. am. am.
  am. am. am.

  app surjective_pr6. app af_function. simpl.
  ir. rwi inc_quotient H3. nin H3. clear H3. nin H4.
  rwi substrate_graph_ea H3. simpl in H3.
  assert(inc (intersection2 (rep y) a) (powerset a)). app powerset_inc.
  app intersection2sub_second.
  cp (powerset_sub H5).
  exists (intersection2 (rep y) a). split. am. aw.
  set_extens. rwi H4 H7. bwi H7. ufi intersection_with H7.
  rwi related_ea H7. ee. simpl in H8. awi H9. Ztac. am. am. am. am. am.
  red in H1. nin H1; am. Ztac. rw H4. bw. rw related_ea.
  uf intersection_with. split. am. split. am.
  aw. sy. am. am. red in H1; nin H1; am. am. am.
Qed.
```

4. *Let* G *be the graph of an equivalence on a set* E*. Show that if* A *is a graph such that* $A \subset G$ *and* $\mathrm{pr}_1 A = E$ *(resp.* $\mathrm{pr}_2 A = E$*) then* $G \circ A = G$ *(resp.* $A \circ G = G$*); furthermore, if* B *is any graph, we have* $(G \cap B) \circ A = G \cap (B \circ A)$ *(resp* $A \circ (G \cap B) = G \cap (A \circ B)$*).*

```
Lemma exercise6_4: forall g a b x,
  let comp := compose_graph in let inter:= intersection2 in
  is_equivalence g -> is_graph a -> is_graph b -> substrate g = x -> sub a g ->
  (domain a = x -> comp g a = g &
  range a = x -> comp a g = g &
  (domain a = x -> comp (inter g b) a = inter g (comp b a)) &
  (range a = x -> comp a (inter g b)  = inter g (comp a b))).
Proof. ir. assert (Ha: is_graph g). fprops.
  split. ir. uf comp. set_extens. awi H5. nin H5. nin H6. nin H6.
  cp (H3 _ H6). assert (J (P x0) (Q x0) = x0). aw. wr H9. equiv_tac.
  cp (Ha _ H5). assert (J (P x0) (Q x0) = x0). aw.
  assert (inc (P x0) (domain a)). rw H4. wr H2. substr_tac.
  awi H8. nin H8. aw. split. am. exists x1. split. am.
  cp (H3 _ H8). wri H7 H5.
  assert (inc (J x1 (P x0)) g).  equiv_tac. equiv_tac. am.
```

Second claim.

```
  split. ir. uf comp. set_extens. awi H5. nin H5. nin H6. nin H6.
  cp (H3 _ H7). assert (inc (J (P x0) (Q x0)) g). equiv_tac. awi H9. am. am.
  assert (J (P x0) (Q x0) = x0). aw. app Ha.
  assert (inc (Q x0) (range a)). rw H4. wr H2.  substr_tac.
  awi H7. nin H7. aw. split. app Ha. exists x1. split. cp (H3 _ H7).
  wri H6 H5. assert (inc (J (Q x0) x1) g). equiv_tac. equiv_tac. am. am.
```

Third claim.

```
  split. ir. uf comp; uf inter. set_extens. awi H5. nin H5. nin H6. nin H6.
  nin (intersection2_both H7). assert (J (P x0) (Q x0) = x0). aw.
  wr H10. app intersection2_inc. cp (H3 _ H6). equiv_tac.
  aw. split. am. exists x1. intuition.
  nin (intersection2_both H5).  assert (J (P x0) (Q x0) = x0). aw. app Ha.
  assert (inc (P x0) (domain a)). rw H4. wr H2. substr_tac.
  wri H8 H6. awi H7. nin H7.  nin H10.  nin H10. aw. split. am. ex_tac.
  app intersection2_inc. cp (H3 _ H10).
  assert (inc (J x1 (P x0)) g). equiv_tac.  equiv_tac.
```

Last claim.

```
  ir. uf comp. uf inter. set_extens. awi H5. nin H5. nin H6. nin H6.
  nin (intersection2_both H6).
  assert (J(P x0) (Q x0) = x0). aw. app intersection2_inc.
  wr H10. cp (H3 _ H7).  equiv_tac. aw.  split. am. ex_tac.
  nin (intersection2_both H5).
  assert (is_pair x0). app Ha. awi H7. nin H7. nin H9. nin H9.
  aw. split. am. exists x1. split. app intersection2_inc.
  cp (H3 _ H10). assert (inc (J (Q x0) x1) g). equiv_tac.
  assert (J(P x0) (Q x0) = x0). aw. wri H13 H6. equiv_tac. am.
Qed.
```

5. *Show that every intersection of graphs of equivalences on a set* E *is the graph of an equivalence on* E. *Give an example of two equivalences on a set* E *such that the union of their graphs is not the graph of an equivalence on* E.

We have already shown the first property. Let's show that the union of two symmetric relations is symmetric.

```
Lemma symmetric_union: forall a b, is_symmetric a -> is_symmetric b ->
  is_symmetric (union2 a b).
Proof. red. ir. red in H; nin H. red in H0; nin H0.
  split. red. ir. cp (union2_or H3). nin H4. app H. app H0.
  ir. red in H3. cp (union2_or H3). nin H4. red. app union2_first.
  app H1. red. app union2_second. app H2.
Qed.
```

We show here that if $G \subset E \times E$, then the substrate of $G \cup \Delta_E$ is E.

```
Lemma substrate_union_diag: forall x g,
  sub g (coarse x) -> substrate (union2 g (diagonal x)) = x.
Proof. ir. ufi coarse H. assert (is_graph g). red. ir. cp (H _ H0).
  awi H1. ee. am.
  assert(is_graph (union2 g (diagonal x))). red. ir. nin (union2_or H1).
  app H0. rwi inc_diagonal H2. nin H2. am.
  set_extens.  rwi inc_substrate_rw H2. nin H2.
  nin H2. nin (union2_or H2). cp (H _ H3).  awi H4. ee; am.
  rwi inc_pair_diagonal H3. nin H3. am. nin H2.
  nin (union2_or H2). cp (H _ H3).  awi H4. ee; am.
  rwi inc_pair_diagonal H3. nin H3.  ue. am.
  assert (inc (J x0 x0) (union2 g (diagonal x))). app union2_second.
  rw inc_pair_diagonal. au.  assert (x0 = P (J x0 x0)). aw. rw H4. substr_tac.
Qed.
```

If a and b are in E, we can consider $\Delta_E \cup \{(a,b),(b,a)\}$. Its substrate is E.

```
Definition special_equivalence a b x :=
  union2 (doubleton (J a b) (J b a))(diagonal x).
Lemma substrate_special_equivalence:forall a b x,
  inc a x -> inc b x ->  substrate(special_equivalence a b x) = x.
Proof. ir. uf special_equivalence. app substrate_union_diag. red. ir.
  uf coarse. nin (doubleton_or H1); rw H2; fprops.
Qed.
```

We show that this is an equivalence.

```
Lemma special_equivalence_ea:forall a b x,
  inc a x -> inc b x -> is_equivalence(special_equivalence a b x).
Proof. ir.
  assert (is_graph (special_equivalence a b x)). uf special_equivalence.
  red. ir. nin (union2_or H1). nin (doubleton_or H2); rw H3; fprops.
  rwi inc_diagonal H2. nin H2. am.
  app symmetric_transitive_equivalence.
  red. split. am. uf special_equivalence. ir. red in H2. nin (union2_or H2).
  red.  app union2_first.
  nin (doubleton_or H3); [cut (J y x0 = J b a) | cut (J y x0 = J a b)]; ir;
  try (rw H5; fprops);  rw (pr1_injective H4); rw (pr2_injective H4); tv.
  red. app union2_second.
  rwi inc_pair_diagonal H3. rw inc_pair_diagonal. intuition. ue.
  red. split. ir. am. uf related. uf special_equivalence. ir.
  nin (union2_or H2).  nin (doubleton_or H4). rw (pr1_injective H5).
  nin (union2_or H3).  nin (doubleton_or H6). rw (pr2_injective H7).
  app union2_first. fprops. rw (pr2_injective H7).
  app union2_second. rw inc_pair_diagonal. au.
  rwi inc_pair_diagonal H6. nin H6. wr H7. rwi (pr1_injective H5) H2. am.
  rw (pr1_injective H5). nin (union2_or H3). nin (doubleton_or H6).
  rw (pr2_injective H7). app union2_second. rww inc_pair_diagonal. au.
  rw (pr2_injective H7). app union2_first. fprops.
  rwi inc_pair_diagonal H6. nin H6. wr H7.
  rw (pr2_injective H5).  app union2_first.  fprops.
  rwi inc_pair_diagonal H4. nin H4. rww H5.
Qed.
```

If we have two such equivalences with (a,b) and (a,c), transitivity of the union would imply that b and c are related in one of the two graphs. If all three elements are distinct this is not possible.

```
Lemma exercise6_5: let x := three_points in
  let g1:= special_equivalence (Ro tpa) (Ro tpb) x in
    let g2:= special_equivalence (Ro tpa) (Ro tpc) x in
      (is_equivalence g1 & is_equivalence g2 & substrate g1 = x &
        substrate g2 = x &  ~ (is_equivalence (union2 g1 g2))).
Proof. ir.
  split. uf g1. app special_equivalence_ea. uf x. ap R_inc. uf x; ap R_inc.
  split. uf g2. app special_equivalence_ea. uf x. ap R_inc. uf x; ap R_inc.
  split. uf g1; uf x. rww substrate_special_equivalence. ap R_inc. ap R_inc.
  split. uf g2; uf x. rww substrate_special_equivalence. ap R_inc. ap R_inc.
  red. ir.
  assert (related (union2 g1 g2) (Ro tpb) (Ro tpa)). red. app union2_first.
  uf g1. uf special_equivalence. app union2_first. fprops.
  assert (related (union2 g1 g2) (Ro tpa) (Ro tpc)). red. app union2_second.
  uf g2. uf special_equivalence. app union2_first. fprops.
```

```
  assert (related (union2 g1 g2) (Ro tpb) (Ro tpc)). equiv_tac.

  red in H2. nin (union2_or H2). nin (union2_or H3).
  nin (doubleton_or H4). cp (pr1_injective H5).  disc.
  cp (pr2_injective H5). disc.  rwi inc_diagonal H4. nin H4. nin H5.
  awi H6. disc.
  nin (union2_or H3). nin (doubleton_or H4). cp (pr1_injective H5).
  disc. cp (pr2_injective H5). disc.
  rwi inc_diagonal H4. nin H4. nin H5. awi H6. disc.
Qed.
```

6. *Let* G, H *be the graphs of two equivalences on* E. *Then* G∘H *is the graph of an equivalence on* E *if and only if* $G \circ H = H \circ G$. *The graph* G ∘ H *is then the intersection of all the graphs of equivalences on* E *wich contain both* G *and* H.

We show that if $G \circ H$ is an equivalence then $G \circ H = H \circ G$. This uses symmetry.

```
Lemma exercise6_6a: forall G H,
  is_equivalence G -> is_equivalence H ->
  (is_equivalence (compose_graph G H) =
    (compose_graph G H = compose_graph H G)).
Proof. ir. set (K:= compose_graph G H).
  app iff_eq. ir.
  set_extens. assert (is_pair x). ufi K H3. awi H3. ee; am.
  assert (J (P x)(Q x) = x). aw. wri H5 H3.
  assert (inc (J (Q x) (P x)) K). equiv_tac. ufi K H6. awi H6. nin H6. nin H7.
  nin H7. aw. split. am. exists x0. split; equiv_tac.
  awi H3. nin H3. nin H4. nin H4.  assert (J (P x)(Q x) = x). aw.
  wr H6. cut (inc (J (Q x) (P x)) K). ir. equiv_tac. uf K. aw. split.
  fprops. exists x0.  split; equiv_tac.
```

Converse. We use Proposition 1 that says that an equivalence satisfies $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$.

```
  ir.  rwi equivalence_pr H1. rwi equivalence_pr H0.
  rw equivalence_pr. ee. uf K. rw composition_associative.
  ufi K H2. rw H2. wr (composition_associative G G H). rw H0.
  wr composition_associative. rw H2. rw composition_associative. rww H1.
  uf K. rw inverse_compose. wr H4; wr H3. am.
Qed.
```

We show here that if G and H are equivalences on E, then the substrate of G ∘ H is E.

```
Lemma exercise6_6b: forall G H,
  is_equivalence G -> is_equivalence H ->  substrate G = substrate H ->
  substrate (compose_graph G H) = substrate G.
Proof.  ir.
  set_extens. ufi substrate H3. nin (union2_or H3); awi H4;
  nin H4; awi H4; nin H4; fprops; nin H5; nin H5.
  assert (x = P (J x x1)). aw. rw H7. rw H2. substr_tac.
  assert (x = Q (J x1 x)). aw. rw H7. substr_tac.
  assert (related G x x). equiv_tac.
  assert (related H x x). rwi H2 H3. equiv_tac.
  assert (related  (compose_graph G H) x x). red. aw. split. fprops.
  exists x. au. substr_tac.
Qed.
```

We prove that the composition is the smallest equivalence that contains G and H.

```
Lemma exercise6_6c: forall G H,
  is_equivalence G -> is_equivalence H -> substrate G = substrate H ->
  (sub G (compose_graph G H) & sub H (compose_graph G H)
    &forall W, is_equivalence W -> sub G W -> sub H W ->
      sub (compose_graph G H) W).
Proof. ir. assert (is_graph G). fprops.
  assert (is_graph H). fprops.
  assert (is_graph  (compose_graph G H)). ap composition_is_graph.
  split. red. ir. cp (H3 _ H6). assert (J (P x)(Q x) =x). aw.
  wr H8. aw. split. wr H8. fprops. wri H8 H6. ex_tac.
  assert (inc (P x) (substrate H)). wr H2. rwi H8 H6;substr_tac. equiv_tac.
  split. red. ir.  cp (H4 _ H6). assert (J (P x)(Q x) =x). aw.
  wr H8. aw. split. wr H8. fprops. wri H8 H6. ex_tac.
  assert (inc (Q x) (substrate G)). rw H2. rwi H8 H6. substr_tac. equiv_tac.
  ir. red. ir. assert (is_pair x). app H5. assert (J (P x)(Q x) =x). aw.
  wri H11 H9. awi H9. nin H9. nin H12. nin H12.
  cp (H8 _ H12).  cp (H7 _ H13). wr H11. equiv_tac. am.
Qed.
```

We know that the domain of an equivalence is the substrate. We show here that the same is true for the domain.

```
Lemma range_is_substrate: forall g,
  is_equivalence g -> range g = substrate g.
Proof.  ir. uf substrate. set_extens. inter2tac.
  nin (union2_or H0). awi H1. nin  H1. aw. exists x0.
  equiv_tac.  fprops. fprops. am.
Qed.
```

If G is an equivalence on E then $G \subset E \times E$.

```
Lemma sub_coarse: forall g,
  is_equivalence g ->  sub g (coarse (substrate g)).
Proof. ir. assert (is_graph g). red in H; nin H; am. cp (sub_graph_prod H0).
  rwi range_is_substrate H1. rwi domain_is_substrate H1. am. am. am.
Qed.
```

The set of all graphs of equivalences on E is a subset of $\mathfrak{P}(E \times E)$, according to the two previous lemmas. We can consider the intersection of all these equivalences that contain G or H (there is at least one, the coarsest equivalence). The intersection is the smallest.

```
Lemma exercise6_6d: forall G H,
  is_equivalence G -> is_equivalence H -> substrate G = substrate H ->
  compose_graph G H = compose_graph H G ->
  (compose_graph G H) = intersection(Zo (powerset (coarse (substrate G)))
    (fun W =>  is_equivalence W & sub G W & sub H W)).
Proof. ir. set (E:= substrate G). assert (sub G (coarse E)). uf E.
  app sub_coarse. assert (sub H (coarse E)). uf E. rw H2. app sub_coarse.
  cp (exercise6_6c H0 H1 H2).
  set_extens. app intersection_inc. exists (coarse E).
  Ztac. ap powerset_inc. fprops. split. ap equivalence_relation_coarse. au.
  ir. Ztac. ee. cp (H14 _ H10 H11 H12). app H15.
  wri (exercise6_6a H0 H1) H3.
```

```
  ap (intersection_forall  (y:=(compose_graph G H)) H7).
  Ztac. app powerset_inc. uf E. wr (exercise6_6b H0 H1 H2). app sub_coarse.
  split. am. intuition.
Qed.
```

7. *Let* G_0, G_1, H_0, H_1 *be the graphs of four equivalences on a set* E *such that* $G_1 \cap H_0 = G_0 \cap H_1$ *and* $G_1 \circ H_0 = G_0 \circ H_1$. *For each* $x \in E$, *let* R_0 *(resp.* S_0*) be the relation induced on* $G_1(x)$ *(resp.* $H_1(x)$*) by the equivalence relation* $(x,y) \in G_0$ *(resp.* $(x,y) \in H_0$*). Show that there exists a bijection of the quotient set* $G_1(x)/R_0$ *onto the quotient set* $H_1(x)/S_0$. *(if* $A = G_1(x) \cap H_1(x)$, *show that both quotient sets are in one-to-one correspondence with the quotient set of* A *by the equivalence relation induced by* R_0 *on* A; *this relation is equivalent to that induced by* S_0 *on* A*).*

This exercice is missing in the French edition. This is the statement we want to prove.

```
Remark exercise6_7: forall G0 G1 H0 H1 E x,
  is_equivalence G0 -> substrate G0 = E ->
  is_equivalence H0 -> substrate H0 = E ->
  is_equivalence G1 -> substrate G1 = E ->
  is_equivalence H1 -> substrate H1 = E ->
  intersection2 G1 H0 = intersection2 G0 H1 ->
  compose_graph G1 H0 = compose_graph G0 H1 ->
  inc x E -> (
    let G1x := image_by_graph G1 (singleton x) in
      let H1x := image_by_graph H1 (singleton x) in
        let R0 := induced_relation G0 G1x in
          let S0 := induced_relation H0 H1x in
            equipotent (quotient R0) (quotient S0)).
```

Let's start with some properties of these relations.

```
Proof. ir. set (A:= intersection2 G1x H1x).
  set(Ar :=induced_relation R0 A).
  set(As :=induced_relation S0 A).
  assert (Ha: is_graph G0). fprops.
  assert (Hb: is_graph H0). fprops.
  assert (Hc: is_graph G1). fprops.
  assert (Hd: is_graph H1). fprops.
  assert (He:axioms_induced_rel G0 G1x). red. split. am. rw H2. uf G1x. red. ir.
  awi H12. nin H12. nin H12. red in H13. wr H6. app (inc_arg2_substrate H13).
  assert (Hf:axioms_induced_rel H0 H1x). red. split. am. rw H4. uf H1x. red. ir.
  awi H12. nin H12. nin H12. red in H13. wr H8. app (inc_arg2_substrate H13).
  assert (is_equivalence R0). uf R0. app equivalence_induced_rel.
  assert (is_equivalence S0). uf S0. app equivalence_induced_rel.
  assert (forall u, inc u G1x = related G1 x u). ir. uf G1x.  aw.
  app iff_eq. ir. nin H14. nin H14. rwi (singleton_eq H14) H15. am.
  ir.  ex_tac.
  assert (forall u, inc u H1x = related H1 x u).  ir. uf H1x.  aw.
  app iff_eq. ir. nin H15. nin H15. rwi (singleton_eq H15) H16. am.
  ir.  ex_tac.
  assert(forall u v, related R0 u v =
    (related G1 x u & related G1 x v & related G0 u v)).
  ir.  uf R0. rw (related_induced_rel u v He). rw H14; rw H14. tv.
  assert(forall u v, related S0 u v =
    (related H1 x u & related H1 x v & related H0 u v)).
  ir.  uf S0. rw (related_induced_rel u v Hf). rw H15; rw H15. tv.
```

Let's show that R_0 and S_0 induce the same relation on A. This uses $G_1 \cap H_0 = G_0 \cap H_1$.

```
assert(axioms_induced_rel R0 A). red.  split. am. red. ir.
ufi A H18. assert (related R0 x0 x0). rw H16. cp (intersection2_first H18).
ufi G1x H19. awi H19. nin H19. nin H19. rwi (singleton_eq H19) H20.
split. am. split. am. app reflexivity. rw H2. wr H6. substr_tac.
substr_tac.
assert(axioms_induced_rel S0 A). red.  split. am. red. ir.
ufi A H19. assert (related S0 x0 x0). rw H17. cp (intersection2_second H19).
ufi H1x H20. awi H20. nin H20. nin H20. rwi (singleton_eq H20) H21.
split. am. split. am. app reflexivity. rw H4. wr H8.
substr_tac.  substr_tac.
assert (is_equivalence  Ar). uf Ar. app equivalence_induced_rel.
assert (is_equivalence  As). uf As. app equivalence_induced_rel.
assert (forall u, inc u A = (related G1 x u & related H1 x u)).
ir. uf A. uf G1x. uf H1x. app iff_eq. ir. nin (intersection2_both H22).
split. awi H23. nin H23. nin H23. rwi (singleton_eq H23) H25. am.
awi H24. nin H24. nin H24. rwi (singleton_eq H24) H25. am.
ir. nin H22. app intersection2_inc; aw; ex_tac.

assert(forall u v, related Ar u v = (related G1 x u & related G1 x v &
  related H1 x u & related H1 x v  & related G0 u v)).
ir. uf Ar. rw (related_induced_rel u v H18). rw H16. rw H22. rw H22.
app iff_eq. intuition. intuition.
assert(forall u v, related As u v = (related G1 x u & related G1 x v &
  related H1 x u & related H1 x v  & related H0 u v)).
ir. uf As. rw (related_induced_rel u v H19). rw H17. rw H22. rw H22.
app iff_eq. intuition. intuition.
assert (forall u v, related Ar u v = related As u v).
ir. rw H23; rw H24. ap iff_eq. ir. ee;try am.
assert (related H1 u v). apply transitivity_e with x. am.
app symmetricity. am. assert (inc (J u v) (intersection2 G0 H1)).
app intersection2_inc. red. wri H9 H31. app (intersection2_second H31).
ir. ee;try am.  assert (related G1 u v). apply transitivity_e with x. am.
app symmetricity. am. assert (inc (J u v) (intersection2 G0 H1)). wr H9.
app intersection2_inc. red. app (intersection2_first H31).
assert (equipotent (quotient Ar) (quotient As)).
assert (Ar = As). rw graph_extensionality. am. nin H20; am. nin H21;am.
rw H26. ap equipotent_reflexive.
```

If π is the canonical projection onto E/R, and $A \subset E$, we know that the quotient set A/R_A is isomorphism to $\pi\langle A\rangle$. Replacing E by $G_1(x)$ and R by R_0, we see that we must show $\pi\langle A\rangle = \pi\langle G_1(x)\rangle$. It seems possible to construct an example where $G_1(x)$ is not empty but A is empty; case where the previous relation is false. We think that the exercise is wrong, but do not have a counterexample.

```
Abort.
```

8. *Let* E, F *be two sets, let* R *be an equivalence relation on* F*, and let* f *be a mapping of* E *into* F*. If* S *is the equivalence relation which is the inverse image of* R *under* f*, and if* $A = f\langle E\rangle$*, define a canonical bijection of* E/S *onto* A/R_A*.*

The first thing to do is to show that S and R_A are equivalence relations.

```
Lemma exercise6_8: forall f r,
```

```
  is_equivalence r -> is_function f -> target f = substrate r ->
  (exists g, bijective g & source g = quotient (inv_image_relation f r) &
    target g = quotient (induced_relation r (image_of_fun f))).
Proof. ir.  set (s := inv_image_relation f r).
  set (A:= (image_of_fun f)). set (Ra := induced_relation r A).
  assert (iirel_axioms f r). red. intuition. assert (A = range (graph f)).
  uf A. uf image_of_fun. red in H0. nin H0. nin H3. rw  H4.
  rw image_by_graph_domain. tv. fprops.
  assert (is_equivalence s).  uf s. app relation_iirel.
  assert (axioms_induced_rel r A). red. split. am. wr H1. rw H3.
  app range_correspondence. red in H0; nin H0; am.
  assert (is_equivalence Ra). uf Ra.  app equivalence_induced_rel.
```

Let's quote the properties of *class_iirel* and *class_induced_rel*: If X is a class modulo R then $f^{-1}\langle X\rangle$ is a class modulo S (if nonempty) and conversely. Classes for R_A are nonempty sets of the form $A\cap X$ where X is a class for R. If a is a class for S we take X such that $a = f^{-1}\langle X\rangle$, and consider $b = A\cap X$. This gives our function. We can do the reverse operation.

We denote by $f_1(a,X)$ the property $a = f^{-1}\langle X\rangle$, $a\cap A\neq\emptyset$ and $X\in F/R$. We denote by $f_2(a)$ a class that satisfies this property, from which we deduce $f_3(a)$ a class for R_A.

```
 set (f1:= fun x=>  fun y => is_class r y
    & nonempty (intersection2 y A) & x = inv_image_by_fun f y).
  assert(forall x, inc x (quotient s) -> exists y, f1 x y). ir.
  rwi inc_quotient H7. ufi s H7. rwi (class_iirel x H2) H7. nin H7.
  exists x0. uf f1. rw H3. am. am.
  set (f2:= fun x => choose (fun y => f1 x y)).
  assert (forall  x, inc x (quotient s) -> f1 x (f2 x)).
  ir. uf f2. app choose_pr.  app (H7 _ H8).
  set (f3:= fun x => intersection2 (f2 x) A).
  assert (forall x,  inc x (quotient s) -> inc (f3 x) (quotient Ra)).
  ir. uf Ra. cp (H8 _ H9). ufi f1 H10. rw inc_quotient. rw class_induced_rel.
  exists (f2 x). intuition. am. am.
```

It is now obvious to find a function from E/S to A/R_A.

```
  set (g:= BL f3 (quotient s) (quotient Ra)).
  exists g. ee; tv. uf g; app bijective_af_function.
```

Our function is injective. Let $X = f_2(a)$ and $X' = f_2(a')$. From $g(a) = g(a')$ we get $f_3(a) = f_3(a')$, namely $X\cap A = X'\cap A$. This is a nonempty set, it constains an element of the form $f(z)$. We have $a = f^{-1}\langle X\rangle$ and $a' = f^{-1}\langle X\rangle'$. These two classes have a common element z, hence are equal.

```
  ir. ufi f3 H12. cp (H8 _ H10). cp (H8 _ H11).  ufi f1 H13. ufi f1 H14. ee.
  nin H15. assert (inc y (f2 v)). inter2tac.
  assert (inc y (f2 u)). wri H12 H15. inter2tac.
  assert (inc y (range (graph f))). wr H3. inter2tac.
  awi H21. nin H21. assert (inc x u). rw H18. uf inv_image_by_fun. aw.
  ex_tac. assert (inc x v). rw H16.
  uf inv_image_by_fun. aw. ex_tac.
  rwi inc_quotient H10. rwi inc_quotient H11. nin (class_dichot H10 H11).
  am. elim (emptyset_pr (x:=x)).  wr H24. app intersection2_inc. am. am. fprops.
```

Surjectivity is easy. Take $y\in A/R_A$. There is some $x\in F/R$ such that $y = x\cap A$ and we want to find $u\in E/S$ such that $g(u) = x\cap A$, $u = f^{-1}\langle x\rangle$. Define $u = f^{-1}\langle x\rangle$. The construction of

g uses the axiom of choice, so that we must show uniqueness, namely $x = f_2(u)$. This is a consequence of the fact these two classes have a common element.

```
  ir. rwi inc_quotient H10. ufi Ra H10.
  rwi class_induced_rel H10. nin H10. nin H10.
  set (u:= inv_image_by_fun f x).
  assert (inc u (quotient s)). uf s. rw inc_quotient. rw class_iirel.
  exists x. wr H3. intuition. am. am. ex_tac.
  uf f3. cp (H8 _ H12). ufi f1 H13.
  nin H11. nin H11. nin (intersection2_both H11). rwi H3 H16. awi H16.
  nin H16. assert (inc x0 u). uf u. uf inv_image_by_fun. aw. ex_tac.
  nin H13. nin H18.
  rwi H19 H17.  ufi inv_image_by_fun H17. awi H17. nin H17. nin H17.
  assert (x1 = y0). wr (W_pr H0 H16).  wr (W_pr H0 H20). tv.
  nin (class_dichot H10 H13). sy. ue.
  elim (emptyset_pr (x:=y0)). wr H22.  app intersection2_inc. ue.
  fprops. am. am.
Qed.
```

9. *Let* F, G *be two sets, let* R *be an equivalence relation of* F*, let* p *be the canonical mapping of* F *onto* F/R *and let* f *be a surjection of* G *onto* F/R*. Show that there exists a set* E*, a surjection* g *of* E *onto* F *and a surjection* h *of* E *onto* G *such that* $p \circ g = f \circ h$*.*

The set E is the disjoint union of F and G, we write it as $\mathrm{E}_a \cup \mathrm{E}_b$.

```
Lemma exercise6_9: forall F G p f r,
  is_equivalence r ->  F = substrate r -> p = canon_proj r ->
  surjective f -> source f = G -> target f = quotient r ->
  exists E, exists g, exists h,
  (surjective g & surjective h & source g = E & source h = E &  target g = F
    & target h = G & compose p g = compose f h).
Proof. ir. set (a:= TPa). set (b:= TPb).
  assert (Hab: b <> a). uf a; uf b. app two_points_distinctb.
  set (Ea:= product F (singleton a)). set (Eb:=product G (singleton b)).
  set (E:= union2 Ea Eb).
  assert (Ha:is_graph E). uf E; uf Ea; uf Eb. red. ir.
  nin (union2_or H5);  awi H6; nin H6; am.
  assert (Hb:forall x, inc x E -> (Q x =a \/ Q x = b)).
  uf E; uf Ea; uf Eb. ir. nin (union2_or H5); awi H6; ee; au.
  assert (Hc:forall x, inc x G -> inc (W x f) (quotient r)).
  ir. wr H4. app inc_W_target. fct_tac. ue.
  assert (Hd:forall x, inc x G -> inc (rep (W x f)) F). ir. rw H0. fprops.
```

We consider the function g; it is the identity on E_a if we identify E_a with F, so that the image is F. Let $x \in \mathrm{E}_b$; we can identify E_b with G, hence assume $x \in \mathrm{G}$ so that $f(x) \in \mathrm{F/R}$. We define $g(x)$ to be a representative of the class of $f(x)$. This is an element of F. We have $p(g(x)) = f(x)$.

```
  set (gz :=fun z=> Yo (Q z = a) (P z) (rep (W (P z) f))).
  assert (He:forall z, inc z Ea -> gz z = P z). uf Ea.
  ir. awi H5. ee. uf gz. rww Y_if_rw.
  assert (Hf:forall z, inc z Ea -> inc (gz z) F).
  ir. rw He. ufi Ea H5. awi H5. intuition. am.
  assert (Hg:forall z, inc z Eb -> gz z = rep (W (P z) f)). uf Eb. ir.
  awi H5. ee. uf gz. rww Y_if_not_rw.  ue.
```

```
assert (Hh:forall z, inc z Eb -> inc (gz z) F). ir. rw Hg. ap Hd.
ufi Eb H5. awi H5. intuition. am.
assert (transf_axioms gz E F). red.  ir. ufi E H5. nin (union2_or H5).
app Hf. app Hh.
set (g:= BL gz E F).
assert(surjective g). uf g. app surjective_af_function. ir.
assert (inc  (J y a) Ea). uf Ea. fprops.
assert (inc  (J y a) E). uf E. inter2tac.
ex_tac. rw He.  aw. am.
assert(forall x, inc x Eb -> W (W x g) (canon_proj r) = W (P x) f).
ir. uf g. aw. rw Hg. app class_rep. app Hc.
ufi Eb H7. awi H7. nin H7. nin H8. am. am. uf E. inter2tac.
wr H0. app Hh. uf E.  inter2tac.
```

We define now h similarly.

```
set (ha:= fun x => (rep (inv_image_by_fun f(singleton(W x (canon_proj r)))))).
assert (Hi:forall x, inc x F ->
  ha x = rep (inv_image_by_fun f (singleton (class r x)))).
ir. uf ha. aw. ue.
assert (Hj:forall x, inc x F ->
  sub (inv_image_by_fun f (singleton(class r x))) G).
ir. red. ir. ufi inv_image_by_fun H9. awi H9. nin H9. nin H9.
red in H2; nin H2.  wr H3. graph_tac.
assert(Hk:forall x, inc x F ->
  inc (ha x) (inv_image_by_fun f (singleton (class r x)))).
ir. rw Hi. app nonempty_rep.
assert (inc (class r x) (target f)). rw H4. rwi H0 H8. gprops.
nin (surjective_pr2 H2 H9). nin H10. exists x0.
uf inv_image_by_fun. aw. exists (class r x). split. fprops. red. wr H11.
ap defined_lem. fct_tac. am. am.
assert (Hl:forall x, inc x F -> inc (ha x) G). ir. ap (Hj _ H8). app Hk.
set(hz:= fun z=> Yo (Q z = a) (ha (P z)) (P z)).
assert (forall z, inc z E -> inc (hz z) G). uf E; uf Ea; uf Eb.
ir. nin(union2_or H8); awi H9; uf hz; ee; rw H11.
rww Y_if_rw. app Hl. rww Y_if_not_rw.
set(h:=BL hz E G).
assert(surjective h). uf h. app surjective_af_function. ir.
assert (inc (J y b) Eb). uf Eb. fprops.
assert (inc (J y b) E). uf E. inter2tac.
ex_tac. uf h. aw. uf hz.  rw Y_if_not_rw. aw. aw.
assert(forall x, inc x Ea -> W (W x h) f = W (P x) (canon_proj r)).
ir. uf h. aw. uf hz.  ufi Ea H10. awi H10. ee.
rw Y_if_rw. cp (Hk _ H11). ufi inv_image_by_fun H13. awi H13. nin H13.
nin H13. red in H14. red in H2. nin H2. rw (W_pr H2 H14).
inter2tac. am. uf E. inter2tac.
wr H0. ufi Ea H10. awi H10. ee. am.
```

We are now ready to prove the main result.

```
exists E. exists g. exists h. ee;tv.
assert (composable p g). red. split. rw H1. app function_canon_proj. split.
fct_tac. rww H1.  sy. am.
assert (composable f h). red. split. fct_tac.  split. fct_tac. am.
ap funct_extensionality. assert (is_function g). fct_tac. fct_tac.
assert (is_function f). fct_tac. fct_tac. tv. sy. rww H1.
```

The non-obvious point is to show $p(g(x)) = f(h(x))$.

```
  ir. aw. simpl in H13.
  ufi E H13. nin (union2_or H13). rw H10. uf g. aw. rw He. rw H1. aw.
  wr He. wr H0. app Hf. am. am. wr He. wr H0. app Hf. am. am.
  uf h. uf hz.  aw. rw Y_if_not_rw. rw H1. app H7. ufi Eb H14. awi H14. ee.
  rww H16.
Qed.
```

10. *(a) if* $\mathrm{R}\{x, y\}$ *is any relation, then "*$\mathrm{R}\{x, y\}$ *and* $\mathrm{R}\{y, x\}$*" is a symmetric relation. Under what condition is it reflexive on a set* E*?*

**(b) Let* $\mathrm{R}\{x, y\}$ *be a reflexive and symmetric relation on a set* E*. Let* $\mathrm{S}\{x, y\}$ *be the relation "There exists an integer* $n > 0$ *and a sequence* $(x_i)_{0\leq i\leq n}$ *of elements of* E *such that* $x_0 = x$*,* $x_n = y$ *and for each index* i *such that* $0 \leq i < n$*,* $\mathrm{R}\{x_i, x_{i+1}\}$*". Show that* $\mathrm{S}\{x, y\}$ *is an equivalence relation on* E *and that its graph is the smallest of all graphs of equivalences on* E *which contain the graph of* R*. The equivalence classes with respect to* S *are called the connected components of* E *with respect to the relation* R*.*

(c) Let $\mathfrak{F}$ *be the set of subsets* A *of* E *such that for each pair of elements* (y, z) *such that* $y \in \mathrm{A}$ *and* $z \in \mathrm{E} - \mathrm{A}$*, we have " not* $\mathrm{R}\{y, z\}$*". For each* $x \in \mathrm{E}$ *show that the intersection of the sets* $\mathrm{A} \in \mathfrak{F}$ *such that* $x \in \mathrm{A}$ *is the connected component of* x *with respect to the relation* R*.**

Part a is trivial.

```
Section Exercice6_10.
Lemma Exercise6_10_a: forall r:EEP,
  symmetric_r (fun  x y => r x y & r y x).
Proof. ir. red. ir. intuition. Qed.
Lemma exercise6_10_b: forall r E,
  reflexive_r r E -> reflexive_r (fun  x y => r x y & r y x) E.
Proof. uf reflexive_r. ir. rw H. app iff_eq. intuition. intuition.Qed.
```

We consider now a context in which R is reflexive and symmetric on E.

```
Variables (R:EEP) (E:Set).
Variables (A1: reflexive_r R E)(A2: symmetric_r R)
  (A3: forall x y, R x y -> inc x E).
```

Defining the relation S is easy.

```
Inductive chain:Type :=
  chain_pair: Set -> Set -> chain
 | chain_next: Set> -> chain -> chain.
Fixpoint chain_head x :=
  match x with chain_pair u _ => u | chain_next u _ => u end.
Fixpoint chain_tail x :=
  match x with chain_pair _ u => u | chain_next _ u => chain_tail u end.
Fixpoint chained_r x :=
  match x with chain_pair u v => R u v
    | chain_next u v => R u (chain_head v) & chained_r v
 end.
Definition relS x y := exists c:chain,
  chained_r c & chain_head c = x  & chain_tail c = y.
```

For the transitivity, we need to concatenate lists.

```
Fixpoint concat_chain x y {struct x} : chain :=
  match x with chain_pair u _ => chain_next u y
| chain_next u v => chain_next u  (concat_chain  (x:=v) y) end.

Lemma head_concat : forall x y,
  chain_head (concat_chain x y) = chain_head x.
Proof. ir. induction x; tv. Qed.

Lemma tail_concat : forall x y,
  chain_tail (concat_chain x y) = chain_tail y.
Proof. ir. induction x; tv. Qed.

Lemma chained_concat: forall x y,
  chained_r x -> chained_r y -> chain_tail x = chain_head y ->
  chained_r (concat_chain x y).
Proof. ir. induction x. split. wrr H1. am.
  nin H. split. rww head_concat. app IHx.
Qed.
Lemma transitiveS: forall x y z, relS x y -> relS y z -> relS x z.
Proof. ir. nin H.  nin H0. ee. exists (concat_chain x0 x1). split.
  app chained_concat.  ue. split. rww head_concat. rww tail_concat.
Qed.
```

For the symmetry, we need to reverse the list. One way to reverse the list L is to start with an empty list L′, and recursively add the head of L to the head of L′, as long as L is not empty. In this case, L and L′ have at least two elements, this gives some special cases to deal with.

```
Fixpoint reconc_chain (x y:chain) {struct x} :chain:=
  match x with chain_pair u v => chain_next v (chain_next u y)
   | chain_next u v => reconc_chain v (chain_next u y) end.

Lemma tail_reconc: forall x y, chain_tail (reconc_chain x y) = chain_tail y.
Proof. intro x.  induction x; ir; tv. simpl. rww IHx.
Qed.
Lemma head_reconc: forall x y, chain_head (reconc_chain x y) = chain_tail x.
Proof. intro x.  induction x. tv. ir. simpl. app IHx.
Qed.
Lemma chained_reconc: forall x y, chained_r x -> chained_r y ->
  R (chain_head y) (chain_head x) ->  chained_r (reconc_chain x y).
Proof. intro x. induction x; simpl; ir. au. ee;  app IHx. simpl. au.
Qed.
```

We define now the reverse.

```
Fixpoint chain_reverse x:=
  match x with chain_pair u v => chain_pair v u
    | chain_next u v =>
      match v with chain_pair u' v' => chain_next v' (chain_pair u' u)
        | chain_next u' v' => reconc_chain v' (chain_pair u' u)
      end end.

Lemma head_reverse: forall x, chain_head (chain_reverse x) = chain_tail x.
Proof. ir.  induction x. tv.  induction x. tv. simpl. app head_reconc.
Qed.
```

```
Lemma tail_reverse: forall x, chain_tail (chain_reverse x) = chain_head x.
Proof. ir. induction x. tv.  induction x. tv. simpl. rww tail_reconc.
Qed.
Lemma chained_reverse: forall x, chained_r x -> chained_r (chain_reverse x).
Proof. ir. induction x. simpl. simpl in H. app A2. induction x.
  simpl.  simpl in H. split.  app A2. nin H. am. app A2. nin H. am.
  simpl. simpl in H. ee. app chained_reconc. simpl. app A2.
Qed.
Lemma symmetricS:  forall x y, relS x y -> relS y x.
Proof. ir. induction x. simpl. au. induction x; simpl; simpl in H; ee.
  au. au.  app chained_reconc. simpl. au.
Qed.
```

We make use of A3 for the first time here. It says that if x is related by S, it is in E. As a consequence our relation is an equivalence relation and its graph is an equivalence on E.

```
Lemma equivalenceS: equivalence_re relS E.
Proof.  ir. red. split. red. split. red. ir. app symmetricS. red. ir.
  apply transitiveS with y. am. am. red. ir. app iff_eq. ir.
  exists (chain_pair y y). split.  simpl. wrr A1. simpl. intuition.
  ir. red in H. nin H. ee. nin x. simpl in H. simpl in H0. wr H0. app (A3 H).
  simpl in H. simpl in H0. wr H0. nin H. app (A3 H).
Qed.
Definition Sgraph := graph_on relS E.
Lemma equivalence_Sgraph: is_equivalence Sgraph.
Proof. uf Sgraph. cp (equivalence_has_graph0 equivalenceS).
  assert (is_graph (graph_on relS E)). app is_graph_graph_on.
  cp equivalenceS. nin H1.
  app (equivalence_if_has_graph2 H0 H H1).
Qed.
Lemma substrate_Sgraph: substrate Sgraph =E.
Proof. uf Sgraph. app extensionality. app substrate_graph_on. red. ir.
  assert (relS x x).  cp equivalenceS. nin H0. wr H1. am.
  cp equivalenceS. rwi (related_graph_on x x  H1) H0. substr_tac.
Qed.
```

We can now show that this is the smallest relation. If r is an equivalence implied by R, the transitivity says that two elements (in particular head and tail) of a *chained_r* chain are related by r.

```
Lemma S_is_smallest: forall r, is_equivalence r ->
  (forall x y, R x y -> inc (J x y) r) -> sub Sgraph r.
Proof.  ir.
  assert (forall w, chained_r w -> inc (J (chain_head w) (chain_tail w)) r).
  ir. induction w. app H0. nin H1.  pose (H0 _ _ H1). pose (IHw H2).
  equiv_tac.
  red. uf Sgraph. uf graph_on. ir. Ztac. nin H4. nin H4. nin H5.
  assert (J (P x)(Q x) = x).  aw.  awi  H3; ee;am.
  wr H7. wr  H5. wr H6. app H1.
Qed.
```

We define here the set $\mathfrak{F}$ and some set C(x). We have to show that this is the class of x for S.

```
Definition setF:= Zo (powerset E)(fun A => forall y z, inc y A ->
  inc z (complement E A) -> not (R y z)).
Definition connected_comp x := intersection(Zo setF (fun A => inc x A)).
```

We first rewrite the condition on $\mathfrak{F}$, then prove that every element of $\mathfrak{F}$ is stable by S, hence contains equivalence classes. Each equivalence class is in $\mathfrak{F}$. The result is then obvious.

```
Lemma setF_pr: forall A a b,
  inc A setF -> inc a A -> R a b -> inc b A.
Proof. ir. ufi setF H. Ztac. nin (inc_or_not b A). tv. ir.
  assert (inc b (complement E A)). srw. split.
  assert (R b a). app A2. app (A3 H5). am. cp (H3 _ _ H0 H5). contradiction.
Qed.
Lemma setF_pr2: forall A a b,
  inc A setF -> inc a A -> relS a b -> inc b A.
Proof. ir. red in H1. nin H1. ee. wr H3. wri H2 H0. clear H2; clear H3.
  induction x; simpl in *. ap (setF_pr H H0 H1).
  nin H1. cp (setF_pr H H0 H1). app IHx.
Qed.
Lemma setF_pr3: forall A a, inc A setF -> inc a A -> sub (class Sgraph a) A.
Proof. ir. red. ir. bwi H1. ufi Sgraph H1. wri related_graph_on H1.
  ap (setF_pr2 H H0 H1). ap equivalenceS. ap equivalence_Sgraph.
Qed.

Lemma setF_pr4: forall a, inc a E -> inc (class Sgraph a) setF.
Proof. ir. uf setF. cp equivalence_Sgraph. Ztac. ap powerset_inc.
  wr substrate_Sgraph. app sub_class_substrate.
  ir. srwi H2. nin H2. red. ir. app H3. bwi H1. bw.
  assert (related Sgraph y a). equiv_tac.  cp equivalenceS.
  ufi Sgraph H5. wri related_graph_on H5. nin H5. ee.
  assert (related Sgraph z a).
  uf Sgraph. wr related_graph_on. red.
  exists (chain_next z x). split. simpl. rw H7. au. simpl. au. am.
  equiv_tac. am. am.
Qed.

Lemma connected_comp_class: forall x, inc x E ->
  class Sgraph x = connected_comp x.
Proof. ir. set_extens. uf connected_comp. app intersection_inc. exists E. Ztac.
  uf setF. Ztac. app powerset_inc. fprops. ir. srwi H2.
  nin H2. elim H3. am. ir. Ztac. ap (setF_pr3 H2 H3). am.
  ufi connected_comp H0.
  cp equivalence_Sgraph.
  assert (inc  (class Sgraph x)  (Zo setF (fun A => inc x A))). Ztac.
  app setF_pr4.  wri substrate_Sgraph H. bw. equiv_tac.
  ap (intersection_forall H0 H2).
Qed.
```

11. *(a) Let* $\mathrm{R}\{x,y\}$ *be a reflexive and symmetric relation on a set* E. R *is said to be intransitive of order 1 if for any four distinct elements* x, y, z, t *of* E, *the relations* $\mathrm{R}\{x,y\}$, $\mathrm{R}\{x,z\}$, $\mathrm{R}\{x,t\}$, $\mathrm{R}\{y,z\}$ *and* $\mathrm{R}\{y,t\}$ *imply* $\mathrm{R}\{z,t\}$. *A subset* A *of* E *is said to be stable with respect to the relation* R *if* $\mathrm{R}\{x,y\}$ *for all* x *and* y *in* A. *If* a *and* b *are two distinct elements of* E *such that* $\mathrm{R}\{a,b\}$ *show that the set* $\mathrm{C}(a,b)$ *of elements* $x \in \mathrm{E}$ *such that* $\mathrm{R}\{a,x\}$ *and* $\mathrm{R}\{b,x\}$ *is stable and that* $\mathrm{C}(x,y) = \mathrm{C}(a,b)$ *for each pair of distinct elements* x, y *of* $\mathrm{C}(a,b)$. *The sets* $\mathrm{C}(a,b)$ *(for each ordered pair* (a,b) *such that* $\mathrm{R}\{a,b\}$*) and the connected components (Exercise 10) with respect to* R *which consist of a single element are called the constituents of* E *with respect to the relation* R. *Show that the intersection of two distinct constituents of* E *contains at most one element*

and that if A, B, C *are three mutually distinct constituents at least one of the sets* $A \cap B$, $B \cap C$, $C \cap A$ *is empty.*

(b) Conversely, let $(X_\lambda)_{\lambda \in L}$ *be a covering of a set* E *consisting of non-empty subsets of* E *having the following properties: (1) if* λ *and* μ *are two distinct indices,* $X_\lambda \cap X_\mu$ *contains at most one element; (2) if* λ, μ, ν *are three distinct letters, then at least one of the three sets* $X_\lambda \cap X_\mu$, $X_\mu \cap X_\nu$, $X_\nu \cap X_\lambda$ *is empty. Let* $R\{\!\{x, y\}\!\}$ *be the relation "There exists* $\lambda \in L$ *such that* $x \in X_\lambda$ *and* $y \in X_\lambda$*"; show that* R *is reflexive on* E*, symmetric and intransitive of order 1, and that the* X_λ *are the constituents of* E *with respect to* R*.*

*(c) * Similarly, a relation* $R\{\!\{x, y\}\!\}$ *which is reflexive and symmetric on* E *is said to be intransitive of order* $n-3$ *if, for every family* $(x_i)_{1 \le i \le n}$ *of distinct elements of* E*, the relations* $R\{\!\{x_i, x_j\}\!\}$ *for each pair* $(i, j) \neq (n-1, n)$ *imply* $R\{\!\{x_{n-1}, x_n\}\!\}$*. Generalize the results of (a) and (b) to intransitive relations of any order. Show that a relation which is intransitive of order p is also intransitive of order q for all* $q > p$*.**

This is a follow-up to the previous exercise. We still assume that R is reflexive and symmetric on E (i.e., A1, A2 and A3 are assumed). We give a short definition and show that it is equivalent to the long one.

```
Definition intransitive1 := forall x y z t,
  x <> y ->  R x y -> R x z -> R x t -> R y z -> R y t -> R z t.

Lemma intransitive1pr :
  let intransitive_alt:= forall x y z t,
    x <> y -> x <> z -> x <> t -> y <> z -> y <> t -> z <> t ->
    inc x E -> inc y E -> inc z E -> inc t E ->
    R x y -> R x z -> R x t -> R y z -> R y t -> R z t in
    intransitive1 = intransitive_alt.
Proof. uf intransitive1. app iff_eq. ir.
  app (H x y z t H0 H10 H11 H12 H13 H14).
  ir.  nin (equal_or_not x z). ue.
  nin (equal_or_not x t). ue. app A2.
  nin (equal_or_not y z). ue.
  nin (equal_or_not y t). ue. app A2.
  assert (inc z E). cp (A2  H4). app (A3 H10).
  nin (equal_or_not z t). ue. wrr A1.  ue.
  app (H x y z t). app (A3 H1). app (A3 H4). cp (A2 H5). app (A3 H12).
Qed.
```

We now define and study $C(a, b)$.

```
Definition stableR A:= forall a b, inc a A -> inc b A -> R a b.
Definition Cab a b:= Zo E (fun x => R a x & R b x).

Lemma Cab_stable: forall a b, a<> b -> R a b -> intransitive1 ->
  stableR (Cab a b).
Proof. ir. red. ir. ufi Cab H2. Ztac. clear H2. ufi Cab H3. Ztac. ee. red in H1.
  app (H1 a b a0 b0).
Qed.
Lemma Cab_trans: forall a b x y, a<> b -> R a b -> intransitive1 ->
  x<> y -> inc x (Cab a b) -> inc y (Cab a b) -> (Cab a b)= (Cab x y).
Proof. ir. red in H1. ufi Cab H3. Ztac. clear H3.
  ufi Cab H4. Ztac. clear H4.
  set_extens. ufi Cab H4. Ztac. clear H4. uf Cab. Ztac. split.
  app (H1 a b x x0). app (H1 a b y x0).
```

```
  ufi Cab H4. Ztac. clear H4. uf Cab. Ztac. ee.
  app (H1 x y a x0). app (H1 a b x y). app A2. app A2.
  app (H1 x y b x0). app (H1 a b x y). app A2. app A2.
Qed.
```

A constituent is either a C or a connected component that has a single element. Let's characterize these. The non-trivial point here is to show that, if x is related to no other element than itself by R, the same is true for S. Hence, consider a chain from x to y. By symmetry, we have a chain from y to x for which we can use induction (if $y \sim x$, then $x = y$ by symmetry of R and equality; if $y \sim z$ and z is chained to x, we get $z = x$ by induction, hence $y \sim x$ and we proceed as above).

```
Lemma singleton_component: forall A, sub A E ->
  (inc A (quotient Sgraph) & is_singleton A) =
  (exists a, A = singleton a & forall b, R a b -> a = b).
Proof. ir. cp equivalence_Sgraph.  rename H0 into Ha.
  ap iff_eq. ir. ee. nin H1. exists x. split. am. ir.
  assert (related Sgraph x b). uf Sgraph. wr related_graph_on. red.
  exists (chain_pair x b). simpl. intuition. app equivalenceS.
  rwi in_class_related H3. nin H3. ee. assert (A = x0). rwi inc_quotient H0.
  nin (class_dichot H0 H3). am. red in H6. elim (emptyset_pr (x:=x)).
  wr H6. app intersection2_inc. rw H1. fprops. am.
  wri H6 H5; rwi H1 H5; inter2tac. am.
  ir. nin H0. nin H0. split. rw inc_quotient. red.
  assert (inc x E). ap H. rw H0. fprops.
  assert (rep A = x).  assert (nonempty A). exists x. rw H0. fprops.
  cp (nonempty_rep H3). rewrite H0 in H4. cp (singleton_eq H4). ue. rw H3.
  split. am. split. rw substrate_Sgraph.  am.
  set_extens.  bw. rwi H0 H4. rw (singleton_eq H4). wri substrate_Sgraph H2.
  equiv_tac.
  bwi H4.
  assert (related Sgraph x0 x). equiv_tac.
  ufi Sgraph H5. wri related_graph_on H5. nin H5. nin H5. nin H6.
  assert (forall c, chained_r c -> chain_tail c = x -> chain_head c = x). ir.
  induction c; simpl in *. rwi H9 H8. sy. app H1. app A2.
  nin H8. rwi IHc H8.  sy. app H1. app A2. am.
  am. rwi (H8 _ H5 H7) H6. rw H0.  rw H6. fprops. ap equivalenceS. am. am.
  red. exists x. am.
Qed.
```

The intersection of two distinct constituents has at least one element. This is obvious if the constituents are singletons. Consider $C(a,b)$ and $C(a',b')$. Assume that they contain u and v. If these elements are distinct then $C(a,b) = C(u,v) = C(a',b')$.

```
Definition is_constituant A :=
  (exists a, A = singleton a & inc a E & forall b, R a b -> a = b) \/
  (exists a, exists b, A = Cab a b& a<> b & R a b).

Lemma constituant_inter2 : forall A B,
  is_constituant A -> is_constituant B -> intransitive1 ->
  A = B \/ small_set (intersection2 A B).
Proof. ir. red in H; red in H0. nin H. right. nin H. nin H. rw H. red. ir.
  cp (intersection2_first H3). cp (intersection2_first H4).
  rw (singleton_eq H5). inter2tac.
```

```
  nin H0. nin H0. nin H0. right. rw H0. red. ir.
  cp (intersection2_second H3). cp (intersection2_second H4).
  rw (singleton_eq H5). inter2tac.
  nin H. nin H. nin H0. nin H0. nin (equal_or_not A B). left. tv.
  right. red. ir. ee. nin (equal_or_not u v). am.
  nin (intersection2_both H3). nin(intersection2_both H4).
  rwi H H10; rwi H H12. cp (Cab_trans H7 H8 H1 H9 H10 H12).
  rwi H0 H11; rwi H0 H13. cp (Cab_trans H5 H6 H1 H9 H11 H13).
  elim H2. ue. ue. ue.
Qed.
```

Consider the case of the intersection of three distinct constituents A, B and C. If A and B are distinct singletons, there intersection is empty. If A is a component $\{x\}$ and if $x \in \mathrm{C}(a,b)$ then x is related to at least two distinct elements, absurd. Assume $\mathrm{A} = \mathrm{C}(a,b)$, $\mathrm{B} = \mathrm{C}(a',b')$ and $\mathrm{C} = \mathrm{C}(a'',b'')$, where the six points are disjoint and related two by two. We have to show that at least one intersection is empty. However we can construct a set E formed of seven points, these six, and an additional one x. Assume that x is related to all points, so that $x \in \mathrm{A} \cap \mathrm{B} \cap \mathrm{C}$. The French edition of Bourbaki adds a last case: *ou les trois ensembles sont identiques*. Hence the claim is: if A, B, C are three mutually distinct constituents at least one of the sets $\mathrm{A} \cap \mathrm{B}$, $\mathrm{B} \cap \mathrm{C}$, $\mathrm{C} \cap \mathrm{A}$ is empty, or the three sets are identical.

```
Proof. ir. nin (equal_or_not A B). left. tv. right.
  nin (equal_or_not A C). left. tv. right.
  nin (equal_or_not B C). left. tv. right.
  nin H.  nin H. ee. nin H0. nin H0. ee. left. app disjoint_pr.
  ir. rwi H H10. awi H10. rwi H0 H11. awi H11. elim H3. ue. ue.
  wr H10; wrr H11.
  nin H0. nin H0. ee. left.  app disjoint_pr. ir.  rwi H H10. awi H10.
  rwi H10 H11.  rwi H0 H11. ufi Cab H11. Ztac. elim H8.
  ee. wr (H7 _  (A2 H13)). wr (H7 _  (A2 H14)). tv.
  nin H0. nin H0. ee. right. right. nin H1. nin H1. ee. left.
  app disjoint_pr. ir. rwi H0 H10. awi H10. rwi H1 H11. awi H11.
  elim H5. ue. ue. wr H10. ue.
  nin H1. nin H1. left.  app disjoint_pr. ir. rwi H0 H8. awi H8. ee.
  rwi H8 H9. rwi H1 H9. ufi Cab H9. Ztac. ee. elim H10.
  wr (H7 _ (A2 H13)). wr (H7 _ (A2 H14)). tv.
  nin H1. nin H1. right. right. left. ap disjoint_pr. ir.
  ee. rwi H1 H7. awi H7. rwi H7 H6. nin H0. nin H0.  ee. rwi H0 H6.
  ufi Cab H6. Ztac. ee. elim H10.
  wr (H9 _ (A2 H13)). wr (H9 _  (A2 H14)). tv.
```

We assume $\mathrm{A} = \mathrm{C}(x, x_2)$, $\mathrm{B} = \mathrm{C}(x_0, x_3)$ and $\mathrm{C} = \mathrm{C}(x_1, x_4)$. We assume $y \in \mathrm{A} \cap \mathrm{B}$, $y_0 \in \mathrm{A} \cap \mathrm{C}$ and $y_1 \in \mathrm{B} \cap \mathrm{C}$. From the first relation we get $y \in \mathrm{A}$, then $y \in \mathrm{E}$ (these two assumptions are cleared). We also get $\mathrm{R}(x, y)$ and $\mathrm{R}(x_2, y)$. This makes 12 relations.

```
 nin (emptyset_dichot (intersection2 A B)). left. tv. right.
  nin (emptyset_dichot (intersection2 A C)). left. tv. right.
  nin (emptyset_dichot (intersection2 B C)). left. tv. right.
  nin H; nin H0; nin H1. nin H; nin H0; nin H1. nin H; nin H0; nin H1.
  nin H9; nin H10; nin H11. nin H6; nin H7; nin H8.
  nin (intersection2_both H6).
  nin (intersection2_both H7).
  nin (intersection2_both H8).
  rwi H H15; rwi H H17; rwi H0 H16; rwi H0 H19; rwi H1 H18; rwi H1 H20.
```

```
  unfold Cab in *. Ztac. clear H20. Ztac. clear H19.
  Ztac. clear H18. Ztac. clear H17. Ztac. clear H16. Ztac. clear H15. ee.
```

We obtain three more relations by intransitivity: elements y, y_0 and y_1 are related. We first prove that each intersection has at most one point.

```
  assert (small_set (intersection2 A B)). assert (is_constituant A). red.
  right. exists x. exists x2. intuition. assert (is_constituant B). red.
  right. exists x0. exists x3. intuition. nin (constituant_inter2 H33 H34 H2).
  elim H3. tv. am.
  assert (small_set (intersection2 A C)). assert (is_constituant A). red.
  right. exists x. exists x2. intuition. assert (is_constituant C). red.
  right. exists x1. exists x4. intuition. nin (constituant_inter2 H34 H35 H2).
  elim H4. tv. am.
  assert (small_set (intersection2 B C)). assert (is_constituant C). red.
  right. exists x1. exists x4. intuition. assert (is_constituant B). red.
  right. exists x0. exists x3. intuition. nin (constituant_inter2 H36 H35 H2).
  elim H5. tv. am.
  cp (H2 _ _ _ _ H9 H12 H15 H25 H27 H29).
  cp (H2 _ _ _ _ H10 H13 H26 H23 H28 H31).
  cp (H2 _ _ _ _ H11 H14 H22 H24 H32 H30).
```

Since x and x_2 are related to y and y_0, if these two elements are distinct, then they are related to y_1. This implies $y_1 \in$ A. Since A $\cap$ B has a single point, this gives $y_1 = y$. Similarly $y_1 = y_0$. This implies $y = y_0$. Absurd. Finally we get $y = y_0 = y_1$.

```
  assert (y= y0). nin (equal_or_not y y0). am.
  cp (H2 _ _ _ _ H39 H36 (A2 H15) H37 (A2 H25) (A2 H38)).
  cp (H2 _ _ _ _ H39 H36 (A2 H27) H37 (A2 H29) (A2 H38)).
  assert (inc y1 A). rw H. uf Cab. Ztac.
  assert (inc y1 (intersection2 A C)). app intersection2_inc.
  ap (intersection2_second H8). cp (H34 _ _ H43 H7).
  assert (inc y1 (intersection2 A B)). app intersection2_inc.
  ap (intersection2_first H8). cp (H33 _ _ H45 H6).  wrr H46.
  assert (y = y1).  assert (inc y (intersection2 B C)).
  app intersection2_inc. app (intersection2_second H6). rw H39.
  app (intersection2_second H7). cp (H35 _ _ H40 H8). tv.
```

The conclusion is then obvious. We close our section.

```
  split. set_extens. rw (H33 _ _ H41 H6). rww H39. rw (H34 _ _ H41 H7). ue.
  set_extens. rw (H35 _ _ H41 H8). wr H40; ue. rw (H34 _ _ H41 H7).
  wr H39. ue.
End Exercice6_10.
```

We consider now part b. Given an assumption on X and E we define a relation R.

```
Definition exercise6_11b_assumption X E:=
  union X = E
  & (forall A, inc A X -> nonempty A)
  & (forall A B, inc A X -> inc B X -> A = B \/ small_set (intersection2 A B))
  & (forall A B C, inc A X -> inc B X -> inc C X ->
    ( A=B \/ A = C \/ B = C \/ intersection2 A B = emptyset
      \/ intersection2 A C = emptyset
```

```
      \/ intersection2 B C = emptyset
      \/ (intersection2 A B = intersection2 A C &
      intersection2 A B = intersection2 B C))).
Definition exercise6_11b_rel X x y := exists A, inc A X & inc x A & inc y A.
```

We start with trivial facts.

```
Lemma exercise6_11b1: forall E X,
  exercise6_11b_assumption X E -> reflexive_r (exercise6_11b_rel X) E.
Proof. ir. red. ir. app iff_eq. ir. red. red in H. ee. wri H H0.
  nin (union_exists H0). exists x. intuition. ir. red in H0. nin H0.
  nin H0. nin H. wr H. nin H1. ap (union_inc H1 H0).
Qed.
Lemma exercise6_11b2: forall E X,
  exercise6_11b_assumption X E -> symmetric_r (exercise6_11b_rel X).
Proof. ir. red. uf exercise6_11b_rel.  ir. nin H0. exists x0. intuition.
Qed.
```

Let's show intransitivity. We assume the four points distinct. We have $x \in x_0 \cap x_1 \cap x_2$, $y \in x_0 \cap x_3 \cap x_4$, $z \in x_1 \cap x_3$ and $t \in x_2 \cap x_4$. We must show that z and t are in a common set. If one of x_1, x_3 is one of x_2, x_4, the result is obvious. We hence get four inequalities between sets. We know that A $\neq$ B implies that the intersection is empty or a singleton. Hence we get $x_1 \cap x_2 = \{x\}$ and $x_3 \cap x_4 = \{y\}$.

```
Lemma exercise6_11b3: forall E X, exercise6_11b_assumption X E ->
  let R := exercise6_11b_rel X in
    forall  x y z t,
      x <> y ->  x<>z -> x <> t -> y <> z -> y <> t -> z <> t ->
      R x y -> R x z -> R x t -> R y z -> R y t -> R z t.
Proof. ir. unfold R in *.  unfold exercise6_11b_rel in *.
  nin H6; nin H7; nin H8; nin H9; nin H10. ee.
  nin (equal_or_not x1 x2). exists x1. intuition. ue.
  nin (equal_or_not x1 x4). exists x1. intuition. ue.
  nin (equal_or_not x3 x2). exists x3. intuition. ue.
  nin (equal_or_not x3 x4). exists x3. intuition. ue.
  assert (intersection2 x1 x2 = singleton x). set_extens.
  assert (inc x (intersection2 x1 x2)). app intersection2_inc.
  red in H. ee. nin (H28 _ _ H7 H8). contradiction. rw (H30 _ _ H25 H26).
  fprops. rw (singleton_eq H25). app intersection2_inc.
  assert (intersection2 x3 x4 = singleton y). set_extens.
  assert (inc y (intersection2 x3 x4)). app intersection2_inc.
  red in H. ee. nin (H29 _ _ H9 H10). contradiction.
  rw (H31 _ _ H26 H27). fprops. rw (singleton_eq H26). app intersection2_inc.
```

We assume $x_0 = x_1$, and study the consequences. We get $x_0 = x_3$ since y and z are two distinct elements in both sets. The case $x_0 = x_4$ is trivial. Consider $x_2 = x_4$; if this is true, we have $x_0 = x_2$ since x and y are in both sets; the result is trivial. In the other case, we have three distinct sets $x_0 = x_1 = x_3$, x_2 and x_4. The intersections of two of them are nonempty. Since these intersections contain distinct elements x, y, t, the sets must be the same and the result is trivial.

```
  nin (equal_or_not x0 x1).
  assert (x0 = x3).
  assert (inc y (intersection2 x0 x3)). app intersection2_inc.
```

```
assert (inc z (intersection2 x0 x3)). wri H27 H18. app intersection2_inc.
red in H; ee. nin (H31 _ _ H6 H9). am. elim H3. ap (H33 _ _ H28 H29).
nin (equal_or_not x0 x4). exists x0. wri H27 H18.  wri H29 H12. intuition.
nin (equal_or_not x2 x4).
assert (inc x (intersection2 x0 x2)). app intersection2_inc.
assert (inc y (intersection2 x0 x2)). app intersection2_inc. rww H30.
red in H; ee. nin (H34 _ _ H6 H8). exists x3. intuition. wr H28. rww H36.
elim H0. ap (H36 _ _ H31 H32).
wri H27 H21.
red in H. ee. nin (H33 _ _ _ H6 H8 H10). contradiction. nin H34.
contradiction. nin H34. contradiction.
assert (inc x (intersection2 x0 x2)). app intersection2_inc.
assert (inc y (intersection2 x0 x4)). app intersection2_inc.
assert (inc t (intersection2 x2 x4)). app intersection2_inc.
nin H34. rwi H34 H35. elim (emptyset_pr H35). nin H34.
rwi H34 H36. elim (emptyset_pr H36). nin H34. rwi H34 H37.
elim (emptyset_pr H37). exists x3. intuition. wr H28. wri H39 H37.
app (intersection2_first H37).
```

We consider now the case $x_0 \neq x_1$. The intersection of these sets is then $\{x\}$. It implies $x_1 \neq x_3$ for otherwise y would be in $x_0 \cap x_1$. Thus $x_1 \cap x_3 = \{z\}$.

```
ir. assert (intersection2 x0 x1 = singleton x).
assert (inc x (intersection2 x0 x1)). app intersection2_inc.
red in H. ee. nin (H30 _ _ H6 H7). elim H27. tv. red in H32. set_extens.
rw (H32 _ _ H28 H33). fprops. rw (singleton_eq H33). am.
nin (equal_or_not x1 x3). assert (inc y (singleton x)).
wr H28. app intersection2_inc. rww H29. elim H0. rww (singleton_eq H30).
assert (intersection2 x1 x3 = singleton z).
assert (inc z (intersection2 x1 x3)). app intersection2_inc.
red in H. ee. nin (H32 _ _ H7 H9). contradiction. red in H34. set_extens.
rw (H34 _ _ H30 H35). fprops. rw (singleton_eq H35). am.
```

Now we compare x_0 and x_4. Assume first equality.

```
ir.
nin (equal_or_not x0 x4).
assert (x0 = x2).
assert (inc x (intersection2 x0 x2)). app intersection2_inc.
assert (inc t (intersection2 x0 x2)). rw H31. app intersection2_inc.
red in H; ee. nin (H35 _ _ H6 H8). am. cp (H37 _ _ H32 H33). elim H2. am.
nin (equal_or_not x0 x3). ir. exists x0. wri H33 H14.  wri H31 H12. intuition.
assert (inc x (intersection2 x0 x1)). app intersection2_inc.
assert (inc y (intersection2 x0 x3)). app intersection2_inc.
assert (inc z (intersection2 x1 x3)). app intersection2_inc.
nin H. ee. cp (H39 _ _ _ H6 H7 H9).  nin H40. contradiction. nin H40.
elim H33. am. nin H40. elim H29. am. nin H40.
rwi H40 H34. elim (emptyset_pr H34). nin H40. rwi H40 H35.
elim (emptyset_pr H35). nin H40. rwi H40 H36.
elim (emptyset_pr H36). exists x2. intuition. wr H32. wri H42 H36.
app (intersection2_first H36).
```

Here $x_0 \neq x_4$. The intersection is $\{y\}$. From this we get $x_2 \cap x_4 = \{t\}$.

```
ir. assert (intersection2 x0 x4 = singleton y).
assert (inc y (intersection2 x0 x4)). app intersection2_inc.
```

```
  red in H. ee. nin (H34 _ _ H6 H10). elim H31. tv. red in H36. set_extens.
  rw (H36 _ _ H32 H37). fprops. rw (singleton_eq H37). am.
  nin (equal_or_not x2 x4).  assert (inc x (singleton y)).
  wr H32. app intersection2_inc. wrr H33. elim H0. rww (singleton_eq H34).
  ir.  assert (intersection2 x2 x4 = singleton t).
  assert (inc t (intersection2 x2 x4)). app intersection2_inc.
  red in H. ee. nin (H36 _ _ H8 H10). elim H33. tv. red in H38. set_extens.
  rw (H38 _ _ H34 H39). fprops. rw (singleton_eq H39). am.
```

On can prove $x_1 \cap x_4 = x_2 \cap x_3 = \emptyset$ but this relation is helpless. The only remaining pairs of sets are (x_0, x_2) and (x_0, x_3). The case $x_0 = x_2 = x_3$ is trivially excluded. The cases $x_0 \neq x_2$ and $x_0 \neq x_3$ are easy.

```
  nin (equal_or_not x0 x2). nin (equal_or_not x0 x3).
  elim H23. wr H35; wr H36; tv. nin H; ee. cp (H39 _ _ _ H6 H7 H9).
  nin H40. elim H27. am. nin H40. elim H36. am. nin H40. elim H29. am.
  nin H40. assert (inc x emptyset).  wr H40. rw H28. fprops.
  elim (emptyset_pr  H41). nin H40. assert (inc y emptyset). wr H40.
  app intersection2_inc. elim (emptyset_pr H41). nin H40.
  assert (inc z emptyset). wr H40. app intersection2_inc.
  elim (emptyset_pr H41). nin H40.  rwi H28 H41. rwi H30 H41.
  elim H1. app (singleton_inj H41).
  nin H. ee. cp (H38 _ _ _ H6 H8 H10).
  nin H39. elim H35. am. nin H39. elim H31. am. nin H39. elim H33. am.
  nin H39. assert (inc x emptyset).  wr H39.  app intersection2_inc.
  elim (emptyset_pr  H40). nin H39. assert (inc y emptyset). wr H39.
  app intersection2_inc. elim (emptyset_pr H40). nin H39.
  assert (inc t emptyset). wr H39. app intersection2_inc.
  elim (emptyset_pr H40). nin H39.  rwi H39 H40. rwi H32 H40. rwi H34 H40.
  elim H4. app (singleton_inj H40).
Qed.
```

We show now that the elements of X are the constituents. Let $p_1(u)$ the property that u has the form $C(a,b)$, $p_2(u)$ the property that u is a connected component formed of a single element. If u satisfies these conditions, then $u \in X$. We are asked to show the converse. Assume that $u \in X$; if it has at least two elements, it satisfies p_1. Assume that it has a single element x. Assume that there is no other set v containing x; then p_2 is true. Assume now that there is another set v containing x; then p_1 and p_2 are false. (Example: E has two elements a and b, X has two elements $\{a,b\}$ and $\{a\}$). The assumptions on X say: if v and v' are two sets containing x, then the intersection is a singleton. Denote by $p_3(u)$ this condition. It does not imply $u \in X$.

Thus we prove the following.

```
Lemma exercise6_11b4: forall E X, exercise6_11b_assumption X E ->
  let R := exercise6_11b_rel X in
    let p1 := fun u => (exists a, exists b, a<> b & R a b & u =
      Zo E (fun x => R a x & R b x)) in
    let p2:= fun u => (exists x, u = singleton x & inc x E&
      forall y, inc y E -> R x y -> x = y) in
    let p3:= fun u => (exists v, inc v X & u <> v & sub u v & is_singleton u) in
      (forall u, inc u X -> p1 u \/ p2 u \/ p3 u ) &
      (forall u, p1 u -> inc u X) & (forall u, p2 u -> inc u X).
```

We show here that singletons satisfy p_2 or p_3.

```
Proof. ir. split. ir. nin (p_or_not_p (is_singleton u)). ir. right.
  nin (p_or_not_p (p3 u)). ir. right. tv. ir. left. red in H1. nin H1.
  exists x. split. am. split. red in H. ee. wr H. apply union_inc with  u.
  rw H1. fprops.  am. ir. red in H4. red in H4. nin H4.
  nin (equal_or_not = y). tv. ir.
  elim H2. exists x0. ee. am. rw H1. red. ir. elim H5. wri H8 H7.
  inter2tac. rw H1. red. ir. inter2tac. red.
  exists x. tv.
```

Our set u is not empty, hence has an element y. We show here that if it has another another element x, then $p_1(u)$ is satisfied. If x_0 is related to x and y, there exists two sets x_1 that contains y and x_0, and x_2 that contains x and x_0. We want to show $x_0 \in u$. This is clear if $x_1 = u$ or $x_2 = u$. Assume these two pairs distinct. If $x_1 = x_2$, the intersection $x_1 \cap u$ is a singleton, containing x and y, absurd. We can then use property (2).

```
  ir. left. red in H. ee. nin (H2 _ H0). exists y.
  nin (p_or_not_p (exists v, inc v u & v <> y)). ir. nin H6. nin H6.
  exists x. split. auto. split. red. red. exists u. auto. set_extens.
  Ztac. wr H. apply union_inc with u. am. am. split. red. red. exists u.
  intuition. red. red. exists u. intuition. Ztac. ee. nin H10. nin H11. ee.
  nin (equal_or_not x1 u). wrr H16.  nin (equal_or_not x2 u). wrr H17.
  assert (inc x (intersection2 u x2)). app intersection2_inc.
  assert (inc y (intersection2 u x1)). app intersection2_inc.
  nin (equal_or_not x1 x2).
  nin (H3 _ _ H0 H10). elim H16. sy. am. red in H21. wri H20 H18.
  elim H7. app H21. ir. cp (H4 _ _ _ H0 H10 H11). nin H21. elim H16. sy; am.
  nin H21. elim H17. sy; am. nin H21. elim H20. am. nin H21.
  assert (inc y emptyset). wr H21. app intersection2_inc.
  elim (emptyset_pr H22). nin H21. assert (inc x emptyset).  wr H21.

  app intersection2_inc. elim (emptyset_pr H22). nin H21.
  assert (inc x0 emptyset).  wr H21.
  app intersection2_inc. elim (emptyset_pr H22). nin H21.
  nin (H3 _ _ H0 H10). elim H16. sy. am. red in H23. wri H21 H18.
  elim H7. app H23.
```

To finish, we must show that a nonempty set that is not a singleton has at least two elements.

```
  ir. elim H1. red. exists y.  set_extens. nin (equal_or_not x y).
  rw H8. fprops. ir. elim H6. exists x. split; am. inter2tac.
```

We show here that $p_1(u)$ implies $u \in \mathrm{X}$. Consider x and x_0 two distinct elements, and $u = \mathrm{C}(x, x_0)$. The two elements x and x_0 are related, this means that they are in a set x_1. We have $u = x_1$. The proof is the same as above.

```
  split. ir. nin H0. nin H0; nin H0. ee. nin H. ee. red in H1. red in H1.
  nin H1. ee.
  assert (u = x1).  rw H2. set_extens.  Ztac. ee.
  nin H10. nin H10. nin H12. nin H11. nin H11. nin H14.
  nin (equal_or_not x1 x3). rww H16. nin (equal_or_not x1 x4). rww H17.
  assert (inc x (intersection2 x1 x3)). app intersection2_inc.
  assert (inc x0 (intersection2 x1 x4)). app intersection2_inc.
  nin (equal_or_not x3 x4).
  nin (H4 _ _ H1 H10). elim H16. am. red in H21. wri H20 H19.
```

```
  elim H0. app H21. ir. cp (H5 _ _ _ H1 H10 H11). nin H21. elim H16. am.
  nin H21. elim H17. am. nin H21. elim H20. am. nin H21.
  assert (inc x emptyset). wr H21. app intersection2_inc.
  elim (emptyset_pr H22). nin H21. assert (inc x0 emptyset).  wr H21.
  app intersection2_inc. elim (emptyset_pr H22). nin H21.
  assert (inc x2 emptyset).  wr H21.
  app intersection2_inc. elim (emptyset_pr H22). nin H21.
  nin (H4 _ _ H1 H10). elim H16. am. red in H23. wri H21 H19.
  elim H0. app H23.
  Ztac. wr H. apply union_inc with x1. am. am. split. red; red.  exists x1.
  intuition. red. red. exists x1. intuition. rw H8. am.
```

We show that $p_2(u)$ implies $u \in \mathrm{X}$.

```
  intuition. red. red. exists x1. intuition. rw H8. am.
  ir. red in H0. nin H0. ee. red in H. ee. wri H H1. cp (union_exists H1).
  nin H6. nin H6. assert (u = x0). set_extens. rwi H0 H8.
  rw (singleton_eq H8). am. assert (inc x1 E). wr H. apply union_inc with x0.
  am. am. assert (R x x1). red. red. exists x0. intuition. wr (H2 _ H9 H10).
  rw H0. fprops. rww H8.
```

Part c. We do not know how to generalize. The last claim is obvious. Assume R intransitive of order $p-3$, let $q > p$ and consider q distinct elements, which are related (with the exception of x_{q-1} and x_q; discard the $q-p$ first elements. The missing relation is true by intransitivity.

Original Exercises.

```
Lemma exercise3_5: forall g a b, is_graph g ->
  (compose_graph (product a b) g = product (inv_image_by_graph g a) b &
  compose_graph g (product a b)  = product a (image_by_graph g b)).

Lemma exercise4_4b:  forall g x,
  fgraph g -> (forall i, inc i (domain g) -> is_graph (V i g)) ->
  nonempty g -> is_singleton x ->
  image_by_graph(intersectionb g) x =
  intersectionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Lemma exercise4_5:  forall g h,
  fgraph g -> (forall i, inc i (domain g) -> is_graph (V i g)) -> is_graph h->
  (compose_graph (unionb g) h =
     unionb (L(domain g) (fun i=> compose_graph(V i g) h))
     & compose_graph h (unionb g) =
     unionb (L(domain g) (fun i=> compose_graph h (V i g)))).

Lemma exercise4_7:  forall G H K, is_graph G ->is_graph H ->is_graph K ->
  sub(intersection2 (compose_graph H G) K)
     (compose_graph(intersection2 H (compose_graph K (inverse_graph G)))
       (intersection2 G (compose_graph (inverse_graph H) K))).
```

Chapter 9

Summary

9.1 The axioms

We give here the list of all axiom schemes.
S1: If $\boldsymbol{A}$ is a relation in $\mathscr{T}$, the relation ($\boldsymbol{A}$ or $\boldsymbol{A}$) $\Longrightarrow$ $\boldsymbol{A}$ is an axiom of $\mathscr{T}$.
S2: If $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations in $\mathscr{T}$, the relation $\boldsymbol{A} \Longrightarrow$ ($\boldsymbol{A}$ or $\boldsymbol{B}$) is an axiom of $\mathscr{T}$.
S3: If $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations in $\mathscr{T}$, the relation ($\boldsymbol{A}$ or $\boldsymbol{B}$) $\Longrightarrow$ ($\boldsymbol{B}$ or $\boldsymbol{A}$) is an axiom of $\mathscr{T}$.
S4: If $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are relations in $\mathscr{T}$, the relation ($\boldsymbol{A} \Longrightarrow \boldsymbol{B}$) $\Longrightarrow$ (($\boldsymbol{C}$ or $\boldsymbol{A}$) $\Longrightarrow$ ($\boldsymbol{C}$ or $\boldsymbol{B}$)) is an axiom of $\mathscr{T}$.
S5: If $\boldsymbol{R}$ is a relation in $\mathscr{T}$, if $\boldsymbol{T}$ is a term in $\mathscr{T}$, and if $\boldsymbol{x}$ a letter, then the relation $(\boldsymbol{T}|\boldsymbol{x})\boldsymbol{R} \Longrightarrow (\exists \boldsymbol{x})\boldsymbol{R}$ is an axiom.
S6: Let $\boldsymbol{x}$ be a letter, let $\boldsymbol{T}$ and $\boldsymbol{U}$ be terms in $\mathscr{T}$, and let $\boldsymbol{R}\{\boldsymbol{x}\}$ a relation in $\mathscr{T}$; then the relation $(\boldsymbol{T} = \boldsymbol{U}) \Longrightarrow (\boldsymbol{R}\{\boldsymbol{T}\} \Longleftrightarrow \boldsymbol{R}\{\boldsymbol{U}\})$ is an axiom.
S7: If $\boldsymbol{R}$ and $\boldsymbol{S}$ are relations in $\mathscr{T}$, and if $\boldsymbol{x}$ is a letter, then the relation $((\forall \boldsymbol{x})(\boldsymbol{R} \Longleftrightarrow \boldsymbol{S})) \Longrightarrow (\tau_{\boldsymbol{x}}(\boldsymbol{R}) = \tau_{\boldsymbol{x}}(\boldsymbol{S}))$ is an axiom.
S8: Let $\boldsymbol{R}$ be a relation, let $\boldsymbol{x}$ and $\boldsymbol{y}$ be distinct letters, and let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be letters distinct from $\boldsymbol{x}$ and $\boldsymbol{y}$ which do not appear in $\boldsymbol{R}$. Then the relation

$$(\forall \boldsymbol{y})(\exists \boldsymbol{X})(\forall \boldsymbol{x})(\boldsymbol{R} \Longrightarrow (\boldsymbol{x} \in \boldsymbol{X})) \Longrightarrow (\forall \boldsymbol{Y})\,\mathrm{Coll}_{\boldsymbol{x}}((\exists \boldsymbol{y})((\boldsymbol{y} \in \boldsymbol{Y}) \text{ and } \boldsymbol{R}))$$

is an axiom.

The French edition has only four axioms since A3 is a theorem.
A1. $(\forall x)(\forall y)((x \subset y \text{ and } y \subset x) \Longrightarrow (x = y))$.
A2. $(\forall x)(\forall y)\mathrm{Coll}_z(z = x \text{ or } z = y)$.
A3. $(\forall x)(\forall x')(\forall y)(\forall y')(((x, y) = (x', y')) \Longrightarrow (x = x' \text{ and } y = y'))$
A4. $(\forall \mathrm{X})\mathrm{Coll}_{\mathrm{Y}}(\mathrm{Y} \subset \mathrm{X})$.
A5. There exists an infinite set.

9.2 The Zermelo Fraenkel Theory

An alternative to the Bourbaki theory is the Zermelo Fraenkel theory. It has the usual interpretation of the quantifiers $\forall$ and $\exists$, but not the symbol τ, thus is missing a choice function. With the notations of [6] the axioms are

B1. $\forall x \forall y[\forall z(z \in x \Longleftrightarrow z \in y) \Longrightarrow x = y]$ (Axiom of extent, A1).

B0. $\forall x \forall y \exists z \forall t[t \in z \Longleftrightarrow (t = x \text{ or } t = y)]$ (Axiom of the pair, A2).

B2. $\forall x \exists y \forall z [z \in y \iff \exists t (t \in x \text{ and } z \in t)]$ (Axiom of the union).

B3. $\forall x \exists y \forall z [z \in y \iff z \subset x]$ (Axiom of the set of subsets, A4).

B4. $\exists x \exists y [\forall z (z \notin y) \text{ and } y \in x \text{ and } \forall u [u \in x \implies \exists v [v \in x \text{ and } \forall t (t \in v \iff t = u \text{ or } t \in u)]]]$ (Axiom of infinity).

SS. $\forall x_1 \ldots \forall x_k \{\forall x \forall y \forall y' [\mathrm{E}(x, y, x_1, \ldots, x_k) \text{ and } \mathrm{E}(x, y', x_1, \ldots, x_k) \implies y = y'] \implies \forall t \exists w \forall v [v \in w \iff \exists u [u \in t \text{ and } \mathrm{E}(u, v, x_1, \ldots, x_k)]]\}$ (Scheme of Substitution).

SC. $\forall x_1 \ldots \forall x_k \forall x \exists y \forall z [z \in y \iff (z \in x \text{ and } \mathrm{A}(z, x_1, \ldots, x))]$ (Scheme of comprehension).

AC. $\forall a \{[\forall x (x \in a \implies x \neq \emptyset) \text{ and } \forall x \forall y (x \in a \text{ and } y \in a \implies x = y \text{ or } x \cap y = \emptyset)] \implies \exists b \forall x \exists u (x \in a \implies b \cap x = \{u\})\}$ (Axiom of choice).

AF. $\forall x [x \neq \emptyset \implies \exists y (y \in x \text{ and } y \cap x = \emptyset)]$ (Axiom of foundation).

Comments. The Zermelo-Frankel theory consists in axioms B1, B2, B3, B4, and scheme SS. From SS, one can deduce SC and B0. The Zermelo theory consists in B1, B0, B2, B3, B4 and SC. It is a weaker theory. Axiom AF is independent of all other axioms, it excludes some weird sets; it is useful in modeling.

Scheme SS depends on a relation E that takes at least two arguments. Fix all parameters but the first two ones. Assume that $\mathrm{E}(x, y)$ is functional in y (i.e., if $\mathrm{E}(x, y) = \mathrm{E}(x, y')$ implies $y = y'$). Rewrite $\mathrm{E}(x, y)$ as $y = f(x)$. The scheme says that for all t, there is a w containing those v of the form $v = f(u)$ for some $u \in t$. Scheme SC says that for every relation $\mathrm{A}(z)$ (that may depend on other parameters), and for every set x there is a set w containing those $v \in x$ that satisfy A.

Consider now axiom B4. The parameter y has to be zero (a.k.a the empty set), and v has to be $u \cup \{u\}$. Denote this by $\mathrm{S}(u)$. Now B4 says: there exists a set x, containing zero, and such that $u \in x \implies \mathrm{S}(u) \in x$. In part two of this report, we shall define pseudo-ordinals. Then the set of finite pseudo-ordinals (which is also the set of finite cardinals with the definition of [6]) is the smallest set satisfying B4. Thus B4 is equivalent to the existence of this set. This axiom is equivalent to A5 (remember that it asserts existence of an infinite set, where "infinite" is a very complicated expression, since it depends on the addition of cardinals, see part two of this report).

Consider now axiom AC. It says that for every set a, if a is formed of non-empty, mutually disjoint sets, there exists a set b that meets each element of a exactly once. Denote by $f(x)$ the unique element of the intersection of x and b. Then (informally) f is a function such that $f(x) \in x$. More formally, the axiom is equivalent to: for every set A, there exists a function $f : \mathfrak{P}(\mathrm{A}) - \emptyset \to \mathrm{A}$ such that $f(x) \in x$. It is also equivalent to say that a product of non-empty sets is non-empty; it is also equivalent to Zermelo's Theorem (every set can be well-ordered, see part 2). We shall use Zermelo's Theorem in order to show that cardinals are well-ordered. A consequence of this fact is the Cantor-Bernstein theorem: if there is an injection from A into B and an injection of B into A, then there is a bijection of A onto B. But this result is independent of AC.

9.3 Unused theorems

We start with a set of lemmas that existed in the first version, and were removed or modified.

Reasoning by cases. We present here the original version of *by_cases*. It assumes that, for any proofs p and p' of P we have $p = p'$, thus $a(p) = a(p')$. Denote this by $a(\mathrm{P})$, if b is a function, we can define $b(\neg\mathrm{P})$ as the common values of b for all proofs of $\neg$P. We define $\mathscr{C}_{\mathrm{C}}(a,b)(\mathrm{P})$ to be $a(\mathrm{P})$ if P is true, and $b(\mathrm{P})$ otherwise. It satisfies the following properties.

```
Lemma by_cases_exists :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T),
    exists x : T, by_cases_pr a b x.
Lemma by_cases_property :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T),
  by_cases_pr a b (by_cases a b).
Lemma by_cases_unique :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (x : T),
    by_cases_pr a b x -> by_cases a b = x.
Lemma by_cases_if :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (p : P),
    by_cases a b = a p.
Lemma by_cases_if_not :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (q : ~ P),
    by_cases a b = b q.
```

The choose function. Let $p(x)$ a predicate over sets. Let $\mathrm{Q}(p,x)$ be the property that, if there is a y such that $p(y)$, then $p(x)$ is true, and if there is no such y, then x is the emptyset. The first two lemmas say that, if we know that there exists a y, then we can simplify. The last lemma says that for every p there exists an x (either y or the emptyset). The code shown here is a simplification of the code of C. Simpson, in that we removed an auxiliary function. The last argument to *choose* was *Type*, but this is not a set anymore. We use here *nat* instead. In the current version, this has been changed to $\emptyset$.

```
Definition refined_pr (p:EP) (x:E) :=
   (ex p -> p x) & ~(ex p) -> x = emptyset.

Lemma refined_pr_if : forall p x, ex p -> refined_pr p x = p x.
Lemma refined_pr_not : forall p x, ~(ex p) -> refined_pr p x = (x = emptyset).
Lemma exists_refined_pr : forall p, ex (refined_pr p).
Definition choose' := fun X : EP => chooseT X (nonemptyT_intro Type).
Definition choose (p:EP) := choose' (refined_pr p).
```

Complement. The first lemma is used to show the second, that says that if it is not the case that both complements are non-empty, then one complement is non-empty, i.e., one set is included in the other.

```
Lemma show_sub_or_aux: forall b c,
   ~ (sub b c \/ sub c b) -> nonempty (complement c b).
Lemma show_sub_or : forall b c,
   (nonempty (complement b c) -> nonempty (complement c b) -> False) ->
   sub b c \/ sub c b.
Lemma non_nonempty_comp_sub: forall a b,
  ~ nonempty (complement a b) -> sub a b.
```

Pairs. Our definition of a pair makes it a set with two elements. This property is not used anymore. We give here the characteristic properties.

```
Lemma inc_pair1: forall x y, inc (pair_first x y) (J x y).
Lemma inc_pair2: forall x y, inc (pair_second x y) (J x y).
Lemma pair_extensionalitya: forall x y u,
  inc u (J x y) -> u = pair_first x y \/ u = pair_second x y.
```

Cartesian Product. We explain here the initial implementation of the cartesian product; it depended on the module *Bounded*, that has been withdrawn.

Consider two sets X and Y. For every $y \in Y$ we can consider the set of all pairs (x, y) with $x \in X$. We can consider the union of these sets. It is denoted by $X \times Y$. Bourbaki defines the product before the union, but uses the same argument as for the union to show existence of the product.

We consider here the following property: Let a be a set and f a function. We say that z is in the record if z is a pair, say $z = (x, y)$, $x \in a$ and $y \in f(x)$. A *Cartesian_record* holds u and v, where u is of type a, and v of type $f(\mathscr{R}u)$. Given such an object i we can create the pair $(\mathscr{R}u, \mathscr{R}v)$. Let's denote it by $g(i)$. If z is in the record, it is of the form $g(i)$. This shows that *in_record* is bounded.

```
Definition in_record (a : Set) (f : EE) (x : Set) :=
  is_pair x  & inc (P x) a   & inc (Q x) (f (P x)).

Record Cartesian_record (a : Set) (f : EE) : Set :=
  {Cartesian_first : a; Cartesian_second : f (Ro Cartesian_first)}.

Definition recordMap (a : Set) (f : EE) (i : Cartesian_record a f) :=
  J (Ro (Cartesian_first i)) (Ro (Cartesian_second i)).

Lemma in_record_ex : forall (a : Set) (f : EE) (x : Set),
 in_record a f x -> exists i : Cartesian_record a f, recordMap i = x.

Lemma in_record_bounded :
  forall (a : Set) (f : EE), Bounded.axioms (in_record a f).
```

The record $R(a, f)$ of a and f is the set of all pairs (x, y) where $x \in a$ and $y \in f(x)$. Following lemmas are trivial.

```
Definition record a f := Bounded.create (in_record a f).

Lemma record_in :  forall a f x, inc x (record a f) -> in_record a f x.
Lemma record_pr :  forall a f x,
   inc x (record a f) -> (is_pair x & inc (P x) a & inc (Q x) (f (P x))).
Lemma record_inc :  forall a f x, in_record a f x -> inc x (record a f).
Lemma record_pair_pr : forall a f x y,
    inc (J x y) (record a f) -> (inc x a & inc y (f x)).
Lemma record_pair_inc :  forall a f x y,
  inc x a -> inc y (f x) -> inc (J x y) (record a f).
```

A product is just a record where the function is constant.

```
Definition product (a b : Set) := record a (fun x : Set => b).
```

Correspondences In a first implementation, a correspondence was a set, more precisely, a functional graph on a set with three elements, Source, Target and Graph.

```
Definition create x y g:=
  denote Source x (denote Target y (denote Graph g stop)).
Definition like (a:E) := a = create(sourceC a) (targetC a)(graphCiN a).
Definition correspondence m:=
   like m & is_graph (graph m) & sub (domain (graph m)) (source m)
   & sub (range (graph m)) (target m).
```

Other lemmas. Most of these theorems assert that (after simplification) we have $x = x$.

```
Definition elt x y := inc y x.
Lemma elt_inc : forall x y, elt x y = inc y x.
Lemma union2_idempotent: forall u,  union2 u u= u.


Lemma source_compose: forall r' r,  source(compose r' r) = source r.
Lemma target_compose: forall r' r,  target(compose r' r) = target r'.
Lemma graph_compose: forall r' r ,
  graph(compose r' r) = compose_graph(graph r')(graph r).
Lemma target_identity: forall x, target (identity_fun x) = x.
Lemma source_identity: forall x, source (identity_fun x) = x.
Lemma graph_identity: forall x, graph (identity_fun x) = diagonal x.
Lemma source_empty_function: source empty_function = emptyset.
Lemma target_empty_function: target empty_function = emptyset.
Lemma source_restriction: forall f x,
  source (restriction_function f x) = x.
Lemma target_restriction:forall f x,
  target (restriction_function f x) = target f.
Lemma graph_restriction:forall f x,
  graph (restriction_function f x) =  (restr  (graph f) x).
Lemma source_restriction2:forall f x y,  source (restriction2 f x y) = x.
Lemma target_restriction2:forall f x y,  target (restriction2 f x y) = y.
Lemma af_source: forall f a b,  source (BL f a b) = a.
Lemma af_target: forall f a b,  target (BL f a b) = b.
Lemma source_first_proj: forall g, source (first_proj g) =  g.
Lemma source_second_proj: forall g, source (second_proj g) = g.
Lemma target_first_proj: forall g, target (first_proj g) = domain g.
Lemma target_second_proj: forall g, target (second_proj g) = range g.
Lemma source_ci: forall a b , source (canonical_injection a b)=a.
Lemma target_ci: forall a b, target (canonical_injection a b)=b.
Lemma graph_ci:  forall a b, graph (canonical_injection a b)= diagonal a.
Lemma source_diag_app:forall a, source (diagonal_application a) = a.
Lemma target_diag_app:forall a, target (diagonal_application a) = product a a.
Lemma source_inv_graph_canon: forall g, source (inv_graph_canon g) = g.
Lemma target_inv_graph_canon: forall g,
  target (inv_graph_canon g) = (inverse_graph g).
Lemma source_fpf :forall f y,
  source (first_partial_fun f y) = domain (source f).
Lemma source_spf :forall f y,
  source (second_partial_fun f y) = range (source f).
Lemma source_fpfa :forall f,
  source (first_partial_function f) = range (source f).
Lemma source_spfa :forall f,
  source (second_partial_function f) = domain (source f).
Lemma source_fpfb :forall a b c,
  source (first_partial_map a b c) = (set_of_functions (product a b) c).
Lemma source_spfb :forall a b c,
```

```
  source (second_partial_map a b c) = (set_of_functions (product a b) c).
Lemma target_fpf :forall f y,
  target(first_partial_fun f y) = target f.
Lemma target_spf :forall f y,
  target(second_partial_fun f y) = target f.
Lemma target_fpfa :forall f,  target(first_partial_function f) =
  set_of_functions (domain (source f)) (target f).
Lemma target_spfa :forall f,  target(second_partial_function f) =
  set_of_functions (range (source f)) (target f).
Lemma target_fpfb :forall a b c,  target(first_partial_map a b c) =
  set_of_functions b (set_of_functions a c).
Lemma target_spfb :forall a b c,  target(second_partial_map a b c) =
  set_of_functions a (set_of_functions b c).
Lemma source_pri: forall f i, source(pr_i f i) = productb f.
Lemma target_pri: forall f i, target(pr_i f i) = V i f.
Lemma source_prit: forall In (f:In->Set) i,  source(pr_it f i) = producttt f.
Lemma target_prit: forall In (f:In->Set) i,  target(pr_it f (Ro i)) = f i.
Lemma source_product1_canon: forall x a,
  source (product1_canon x a) = x.
Lemma target_product1_canon: forall x a,
  target (product1_canon x a) = (product1 x a).
Lemma source_product2_canon: forall x y,
  source (product2_canon x y) = (product x y).
Lemma target_product2_canon: forall x y,
  target (product2_canon x y) = (product2 x y).
Lemma source_pc: forall f u, source (product_compose f u) = (productb f).
Lemma target_pc: forall f u,
  target (product_compose f u) =
  (productf (source u) (fun k => V (W k u) f)).
Lemma source_prj: forall f j,
  source (pr_j f j ) = (productb f).
Lemma target_prj: forall f j,
  target (pr_j f j ) = restriction_product f j.
Lemma source_pam: forall f g,
  source (prod_assoc_map f g) = productb f.
Lemma target_pam: forall f g,
  target (prod_assoc_map f g) =
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma source_popc:forall f f',
  source (prod_of_products_canon f f') =
  (product (productb f) (productb f')).
Lemma source_ext_map_prod: forall  In src trg f,
  source (ext_map_prod In src trg f) = (productf In src ).
Lemma target_ext_map_prod: forall  In src trg f,
  target (ext_map_prod In src trg f) = (productf In trg).
Lemma source_ext_to_prod: forall  u v,
  source (ext_to_prod u v) = product (source u) (source v).
Lemma target_ext_to_prod: forall u v,
  target (ext_to_prod u v) = product (target u) (target v).

Definition canon_projc f := BL(fun x=> gclass f x)
  (source f) (quotient (graph f)).
Lemma source_canon_proj: forall r,
  source (canon_proj r) = substrate r.
Lemma target_canon_proj: forall r,
  target (canon_proj r) = quotient r.
```

```
Lemma source_canon_projc: forall f,
  source (canon_projc f) = source f.
Lemma target_canon_projc: forall f,
  target (canon_projc f) = quotient (graph f).
Lemma source_section_canon_proj: forall r,
  source (section_canon_proj r) = (quotient r).
Lemma target_section_canon_proj: forall r,
  target (section_canon_proj r) = (substrate r).
Lemma source_foq: forall r f b,
  source (function_on_quotient r f b)=  quotient r.
Lemma source_foqs: forall r r' f,
  source (function_on_quotients r r' f)=  quotient r.
Lemma source_foqc: forall r f,
  source (fun_on_quotient r f)=  quotient r.
Lemma source_foqcs: forall r r' f,
  source (fun_on_quotients r r' f)=  quotient r.
Lemma target_foq: forall r f b,  target (function_on_quotient r f b) =  b.
Lemma target_foqs: forall r r' f ,
   target (function_on_quotients r r' f)=  quotient r'.
Lemma target_foqc: forall r f,  target (fun_on_quotient r f)=  target f.
Lemma target_foqcs: forall r r' f,
  target(fun_on_quotients r r' f)= quotient r'.

Lemma composable_identity_left: forall m,
  is_correspondence m -> composableC (identity_fun (target m)) m.
Lemma composable_identity_right: forall m,
  is_correspondence m -> composableC m  (identity_fun (source m)).
Lemma correspondence_of_restricted_eq: forall x,
  graph_to_eq_cor(diagonal x) = identity_fun x.
Theorem equivalence_cor_pr: forall f,
  is_correspondence f ->
  (equivalence_cor f = (source f = target f &
    (source f = (domain (graph f))) & compose f f = f &
    f = inverse_fun f)).
Lemma restriction_correspondence: forall f x,
  is_function f-> sub x (source f)
  -> is_correspondence(restriction_function f x).
Lemma w_identity: forall a x,  identityC a x = x.
Lemma inc_create_domain: forall sf f a,
  inc a (domain (L sf f))  = inc a sf.
Lemma restriction_V: forall f x i,
  fgraph f -> sub x (domain f) -> inc i x ->  V i (restr f x) = V i f.
```

9.4 Tactics

We start with some abbreviations. They have in general two letters as in *rw*; they have the form *rwi* when they apply to a given hypothesis.

```
Ltac ir := intros.
Ltac rw u := rewrite u.
Ltac rwi u h := rewrite u in h.
Ltac wr u := rewrite <- u.
Ltac wri u h := rewrite <- u in h.
Ltac ap h := apply h.
Ltac om := omega.
```

```
Ltac am := assumption.
Ltac tv := trivial.
Ltac eau := eauto.
Ltac sy := symmetry.
Ltac uf u := unfold u.
Ltac ufi u h := unfold u in h.
Ltac nin h := induction h.
Ltac uh a := red in a.
Ltac au := first [ solve [am] | auto ].
```

Tactics where the last letter is doubled are extensions that call *tv* in order to solve a goal.

```
Ltac app u := ap u; tv.
Ltac rww u := rw u; tv.
Ltac rwii u h:= rwi u h; tv.
Ltac wrr u := wr u; tv.
```

This tactic helps solving $a = b$ by application of the axiom of extent for sets.

```
Ltac set_extens:= app extensionality; unfold sub; intros.
```

We define here a tactic *ee* that removes all conjunctions.

```
Ltac EasyDeconj :=
  match goal with
  |  |- (_ & _) => ap conj; [ EasyDeconj | EasyDeconj ]
  |  |- _ => idtac
  end.
Ltac EasyExpand :=
  match goal with
  | id1:(?X1 /\ ?X2) |- _ => nin id1; EasyExpand
  |  |- _ => EasyDeconj
  end.
Ltac ee := EasyExpand.
```

The *cp* tactic is a variant of *set* or *pose*.

```
Ltac Remind u :=
  set (recalx := u);
   match goal with
   | recalx:?X1 |- _ => assert X1; [ exact recalx | clear recalx ]
   end.
Ltac cp := Remind.
```

The *Ztac* tactic can be used when the assumption or conclusion (checked in this order) contains $x \in \{y \in z | \mathrm{P}(y)\}$; it replaces by: $x \in z$ and $\mathrm{P}(x)$.

```
Ltac Ztac :=
  match goal with
  | id1:(inc ?X1 (Zo _ _)) |- _ => nin (Z_all id1)
  |  |- (inc _ (Zo _ _)) => ap Z_inc; auto
  | _ => idtac
  end.
```

The *fprops* tactic applies lemmas from the *fprops* data base. The tactics *aw* and *srw* apply rewrite rules from the *aw* and *sw* data base. The tactics *awi* and *srwi* apply these rules in a given hypothesis, rather than the conclusion.

```
Ltac aw := autorewrite with aw;  tv.
Ltac awi u:= autorewrite with aw in u.
Ltac awii u:= autorewrite with aw in u; tv.
Ltac bw := autorewrite with bw; tv.
Ltac bwi u:= autorewrite with bw in u.
Ltac srw:= autorewrite with sw; tv.
Ltac srwi u:= autorewrite with sw in u.

Ltac fprops := auto with fprops.
Ltac gprops := auto with gprops.
```

The *dneg* tactic implements the rule: if A $\Longrightarrow$ B then $\neg$B $\Longrightarrow$ $\neg$A.

```
Ltac dneg := match goal with
  H : ~ _ |- ~ _ => red; ir; elim H
end.
```

The next tactics have been introduced in Version 3. We were able to reduce by 17% teh size of the file *sete2.v* described in the previous chapter.

The next tactic rewrites equation $a = b$ in some cases.

```
Ltac Use_eq :=
  match goal with
    | H:?a = ?b |- ?f ?a = ?f ?b => rww H
    | H:?a = ?b |- ?f ?a _ = ?f ?b _ => rww H
    | H:?a = ?b |- ?f _ ?a = ?f _ ?b  => rww H
    | H:?a = ?b |- ?f ?a _ _  = ?f ?b _ _  => rww H
    | H:?a = ?b |- ?f _ ?a _   = ?f _  ?b _  => rww H
    | H:?a = ?b |- ?f _ _ ?a  = ?f _ _ ?b  => rww H
    | H:?a = ?b |- _ => solve [ rww  H ; fprops | wrr H ; fprops]
    | Ha : ?a = ?c, Hb : ?b =  ?c |- ?a = ?b => wr Ha ; wr Hb; tv
    | H:?b = ?a |- _ ?b => rww H
    | H:?b = ?a |- _ _ ?b  =>  rww H
    | H:?b = ?a |- _ ?b _ =>  rww H
    | H:?b = ?a |- _ _ ?b  =>  rww H
    | H:?b = ?a |- _ ?b _ _ =>  rww H
    | H:?b = ?a |- _ _ _ ?b  =>  rww H
    | H:?b = ?a |- _ _ ?b _ =>  rww H
end.
```

This tactics solves goals involving existence and pairs.

```
Ltac ex_tac:=
  match goal with
   | H:inc (J ?x ?y) ?z |- exists x, inc (J x ?y) ?z
     => exists x ; am
   | H:inc (J ?x ?y) ?z |- exists y, inc (J ?x y) ?z
     => exists y ; am
   | |- exists y, inc (J (P ?x) y) _
     => exists (Q x) ;aw
   | |- exists y, inc (J y (Q ?x)) _
```

```
    => exists (P x) ; aw
  | H:inc (J ?x ?y) ?z |- exists x, _ & inc (J x ?y) ?z
    => exists x ; split; tv
  | H:inc (J ?x ?y) ?z |- exists y, _ & inc (J ?x y) ?z
    => exists y ; split; tv
  | H:inc (J ?x ?y) ?z |- exists x, inc (J x ?y) ?z & _
    => exists x ; split; tv
  | H:inc (J ?x ?y) ?z |- exists y, inc (J ?x y) ?z & _
    => exists y ; split; tv
  | H:inc ?x ?y |- exists x, inc x ?y & _
    => exists x ; split; tv
  |  |- exists x, inc x (singleton ?y) & _
    => exists y ; split; fprops
  | H : inc (J ?x ?y) ?g |-  inc ?x (domain ?g)
      => aw; exists y ;am
  | H : inc (J ?x ?y) ?g |-  inc ?y (range ?g)
      => aw; exists x;am
  | H : inc ?x ?y |-  nonempty ?y
      => exists x;am
end.
```

The *inter2tac* tactic helps solvings goals of the form $x \in A \cup B$ or consequences of $x \in A \cap B$.

```
Ltac inter2tac :=
  match goal with
    | H:inc ?X1 (intersection2 ?X2 _ ) |- inc ?X1 ?X2
      => ap (intersection2_first H)
    | H:inc ?X1 (intersection2 _ ?X2 ) |- inc ?X1 ?X2
      => ap (intersection2_second H)
    | H:inc ?X1 ?X2 |- inc ?X1 (union2 ?X2 _)
      => app union2_first
    | H:inc ?X1 ?X2 |- inc ?X1 (union2 _ ?X2)
      => app union2_second
    |   |- inc _ (intersection2 _ _ )
      => app intersection2_inc
    | H:J _ _ = J _ _ |- _
       => solve [ rww (pr1_injective H);fprops | rww (pr2_injective H) ; fprops]
    | H: inc _ (singleton  _ ) |- _
      => solve [ rww (singleton_eq H);fprops ]
  end.
```

The next tactic solves a goal of the form: f is a function.

```
Ltac fct_tac :=
  match goal with
    | H:bijective ?X1 |- is_function ?X1 => exact (bij_is_function H)
    | H:injective ?X1 |- is_function ?X1 => exact (inj_is_function H)
    | H:surjective ?X1 |- is_function ?X1 => exact (surj_is_function H)
    | H:is_function ?X1 |- is_correspondence ?X1 => nin H; am
    | H:composable ?X1 _ |- is_function ?X1 => destruct H as [H _ ]; exact H
    | H:composable _ ?X1 |- is_function ?X1 => destruct H as [_ [H _ ]]; exact H
    | H:composable ?f ?g |- is_function (compose ?f ?g ) =>
     ap (is_function_compose H)
    | H:is_function ?f |- is_function (compose ?f ?g ) =>
     ap is_function_compose; red; ee; tv
    | H:is_function ?g |- is_function (compose ?f ?g ) =>
```

```
    ap is_function_compose; red; ee; tv
  end.
```

This solves a goal related to a graph of a function.

```
Ltac graph_tac :=
  match goal with
    | Ha:is_function ?X1, Hb: inc (J _ ?X2) (graph ?X1)
      |-  inc ?X2 (target ?X1)
        => ap (inc_pr2graph_target Ha Hb)
    | Ha:is_function ?X1, Hb: inc (J ?X2 _) (graph ?X1)
      |-  inc ?X2 (source ?X1)
        => ap (inc_pr1graph_source Ha Hb)
    | Ha:is_function ?X1, Hb: inc ?X2 (graph ?X1)
      |-  inc (P ?X2) (source ?X1)
        => ap (inc_pr1graph_source1 Ha Hb)
    | |- inc (J ?x (W ?x ?f)) (graph ?f) => app defined_lem
  end.
```

This helps solving equality of functions.

```
Ltac corr_tac :=
  match goal with
    | H: corr_value _ = corr_value _ |- _
      => rwi correspondence_extensionality1 H
    | Ha: corr_value ?a = ?b, Hb:corr_value ?c = ?d |- ?b = ?d
      => wr Ha; wr Hb ;rww correspondence_extensionality1
end.
```

This helps solving goals that depends on the substrate of a relation.

```
Ltac substr_tac :=
  match  goal with
    | H:inc ?x ?r |- inc (P ?x) (substrate ?r) => ap (inc_pr1_substrate H)
    | H:inc ?x ?r |- inc (Q ?x) (substrate ?r) => ap (inc_pr2_substrate H)
    | H:related ?r ?x ?y |- inc ?x (substrate ?r) => ap (inc_arg1_substrate H)
    | H:related ?r ?x ?y |- inc ?y (substrate ?r) => ap (inc_arg2_substrate H)
  end.
```

This tatics exploits the properties of an equivalence relation.

```
Ltac equiv_tac:=
  match goal with
    | H: is_equivalence ?r, H1: inc ?u (substrate ?r) |- related ?r ?u ?u
      => ap (reflexivity_e H H1)
    | H: is_equivalence ?r |- inc (J ?u ?u) ?r
      => app reflexivity_e
    | H:is_equivalence ?r, H1:related ?r ?u ?v |- related ?r ?v ?u
      => app (symmetricity H H1)
    | H:is_equivalence ?r, H1: inc (J ?u ?v) ?r |- inc (J ?v ?u) ?r
      => app (symmetricity H H1)
    | H:is_equivalence ?r, H1:related ?r ?u ?v, H2: related ?r ?v ?w
      |- related ?r ?u ?w
      => app (transitivity_e H H1 H2)
    | H:is_equivalence ?r, H1:related ?r ?v ?u, H2: related ?r ?v ?w
```

```
      |- related ?r ?u ?w
      => app (transitivity_e H (symmetricity H H1) H2)
    | H: is_equivalence ?r, H1: inc (J ?u ?v) ?r, H2: inc (J ?v ?w) ?r |-
      inc (J ?u ?w) ?r
      => app (transitivity_e H H1 H2)
end.
```

This tactic solves goals of the form $x \in \bigcup_{i\in \mathrm{I}} \mathrm{X}_i$, by guessing the value of i.

```
Ltac union_tac:=
  match goal with
    | H:inc ?x (?f ?y) |- inc ?x (uniont ?f)
      => ap (uniont_inc f y H)
    | Ha : inc ?i (domain ?g), Hb :  inc ?x (V ?i ?g) |- inc ?x (unionb ?g)
      => ap (unionb_inc Ha Hb)
    | Ha : inc ?x ?y, Hb :  inc ?y ?a |- inc ?x (union ?a)
      => ap (union_inc Ha Hb)
    | Ha : inc ?y ?i, Hb :  inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => ap (unionf_inc _ Ha Hb)
    | Ha : inc ?y ?i |- inc ?x (unionf ?i ?f)
      => apply unionf_inc with y; fprops
    | Ha : inc ?i (domain ?g) |- inc ?x (unionb ?g)
      => apply unionb_inc with i; fprops
    | Ha : inc ?x (V ?i ?g) |- inc ?x (unionb ?g)
      => apply unionb_inc with i; fprops
    | Hb : inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => apply unionf_inc with y; fprops
  end.
```

This tactic replaces a goal $x = \emptyset$ by a goal *False* undert the assumption $y \in x$.

```
Ltac empty_tac :=
  match goal with
  |  |-  _ = emptyset => ap is_emptyset; red;ir; idtac
  | _ => idtac
  end.
```

9.5 Rewriting rules

The *srw* and *srwi* tactics use the following rewriting rules [1]: double_complement, inc_complement, empty_product2, empty_product1.

The *aw* and *awi* tactics use the following rewriting rules: pr1_pair, pr2_pair, fun_image_rw, range_pr, domain_pr, create_domain, create_V_rewrite, range_emptyset, domain_emptyset, inc_pair_diagonal, restriction_graph_pr, image_by_graph_pr, cut_pr, inverse_graph_pair, inverse_graph_emptyset, inverse_diagonal, inverse_product, inv_image_graph_pr, inc_compose W_af_function, source_compose, target_compose, compose_related, graph_compose, W_compose, pair_recov, W_canon_proj, source_canon_proj target_canon_proj.

The tactic *fprops* uses the following lemmas: sub_refl, singleton_inc, pair_is_pair, doubleton_first, doubleton_second, fcompose_axioms, nonempty_singleton,

[1] In version 2, we added theorems to the lists shown here

product_is_graph, emptyset_is_graph, diagonal_is_graph, range_correspondence, domain_correspondence, is_graph_correspondence, restricted_graph_is_graph, inverse_graph_is_graph, correspondence_inverse_fun, composition_is_graph, fgraph_function is_graph_function, function_acreate, product_pair_inc, inc_W_target, function_canon_proj, inc_rep_substrate.

9.6 List of Theorems

We give here the list of all theorems, propositions, lemmas, corollaries, together with the Coq names, a page reference, and the statement (we use French quotes for exact citations).

Section one

Proposition 1 (*sub_refl*) « $x \subset x$ », [17].

Proposition 2 (*sub_trans*) « $(x \subset y$ and $y \subset z) \implies (x \subset z)$ », [17].

Theorem 1 « The relation $(\forall x)(x \notin \mathrm{X})$ is functional in X. » This theorem asserts existence and uniqueness of the empty set, [17].

Section 2

Theorem 1 asserts existence of the product $\mathrm{X} \times \mathrm{Y}$ of two sets, [29].

Proposition 1 (*product_monotone* and variants) « If A', B' are non-empty sets, the relation $\mathrm{A}' \times \mathrm{B}' \subset \mathrm{A} \times \mathrm{B}$ is equivalent to "$\mathrm{A}' \subset \mathrm{A}$ and $\mathrm{B}' \subset \mathrm{B}$" », [29].

Proposition 2 (*empty_product_pr*) « Let A and B be two sets. The relation $\mathrm{A} \times \mathrm{B} = \emptyset$ is equivalent to "$\mathrm{A} = \emptyset$ or $\mathrm{B} = \emptyset$" », [29]

Section 3

Proposition 1 (*range_domain_exists*) asserts existence and uniqueness of the range and domain of a graph, [42].

Proposition 2 (*image_by_increasing*) « Let G be a graph and let X, Y be two sets; then the relation $\mathrm{X} \subset \mathrm{Y}$ implies $\mathrm{G}\langle \mathrm{X} \rangle \subset \mathrm{G}\langle \mathrm{Y} \rangle$ », [45].

Corollary (*image_of_large*).

Proposition 3 (*inverse_compose*) « Let G, G' be two graphs. The inverse of $\mathrm{G}' \circ \mathrm{G}$ is then $\overset{-1}{\mathrm{G}} \circ \overset{-1}{\mathrm{G}'}$ », [47].

Proposition 4 (*composition_associative*) is associativity of composition of graphs, [47].

Proposition 5 (*image_composition*) says $(\mathrm{G}' \circ \mathrm{G})\langle \mathrm{A} \rangle = \mathrm{G}'\langle \mathrm{G}\langle \mathrm{A} \rangle \rangle$, [47].

Proposition 6 (*is_function_compose*) says « If f is a mapping of A into B and g is a mapping of B into C, then $g \circ f$ is a mapping of A into C », [82].

Proposition 7 (*bijective_inv_function* and *inv_function_bijective*) says « Let f be a mapping of A into B. Then $\overset{-1}{f}$ is a function if and only if f is bijective », [63].

Proposition 8 (*inj_if_exists_left_inv*, and variants) says under which conditions a function has a left or right inverse, [66].

Corollary (*bijective_from_compose*).

Theorem 1 (*inj_compose*) and variants) studies the relationship between injectivity, surjectivity and composition, [68].

Proposition 9 (*exists_left_composable* and variants) explains when a function can be factored through another one, [70].

Section 4

Proposition 1 (*uniont_rewrite*, *intersectiont_rewrite* and variants) says that if $f : K \to I$ is a function, $(X_\iota)_{\iota\in I}$ a family of sets, then the union and the intersection of the family is the union and the intersection of $X_{f(\kappa)}$ over K, [78].

Proposition 2 (*union_assoc* and *intersection_assoc*) states associativity of union and intersection, [79].

Proposition 3 (*image_of_union* and *image_of_intersection*) says that if Γ is a correspondence, $\Gamma\langle\bigcup X_\iota\rangle = \bigcup\Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap\Gamma\langle X_\iota\rangle$, [79].

Proposition 4 (*inv_image_of_intersection*) says equality holds for the inverse image of intersection, [80].

Corollary (*inj_image_of_intersection*).

Proposition 5 (*complementary_union* and *complementary_intersection*) studies the complementary of unions and intersections, [80].

Proposition 6 (*inv_image_of_comp*) studies the inverse image of the complementary, [82].

Corollary (*inj_image_of_comp*).

Proposition 7 (*agrees_on_covering* and *prolongation_covering*) says that if X_ι is a covering of E, then two functions that agree on each X_ι agree on E, and a function defined on each X_ι can be extended to E if the obvious compatibility conditions hold, [83].

Proposition 8 (*prolongation_partition*) says that if $(X_\iota)_\iota$ is a partition of X and $f_\iota \in \mathscr{F}(X_\iota, T)$, then there exists a unique $f \in \mathscr{F}(X, T)$ that extends every f_ι , [86].

Proposition 9 (*disjoint_union_lemma*) asserts existence of the disjoint union, [87].

Proposition 10 (*disjoint_union_pr*) relates sum and union, [87].

9.6.1 Section 5

Proposition 1 (*surjective_etp* and *injective_etp*) says: if f is surjective (resp. injective), then its extension to the set of sets is surjective (resp. injective), [89].

Proposition 2 (*injective_c3f* and *surjective_c3f*) states under which conditions $f \mapsto v \circ f \circ u$ is injective or surjective, [91].

Corollary (*bijective_c3f*).

Proposition 3 (*bijective_fpfa* and *bijective_spfa*) says that $\mathscr{F}(B \times C; A)$, $\mathscr{F}(B; \mathscr{F}(C; A)$ and $\mathscr{F}(C; \mathscr{F}(B; A))$ are canonically isomorphic, [92].

Proposition 4 (*bijective_pc*) says: Given a family X_ι and a bijection f, the product $\prod X_\iota$ is isomorphic to the product $\prod X_{f(\iota)}$, [98].

Propositions 6 and 5 (*prolongation_exists* and *surjective_prj*) if X_ι is nonempty for $\iota \notin J$, then pr_J is surjective from the product $\prod_{\iota\in I} X_\iota$ into the partial product $\prod_{\iota\in J} X_\iota$ [99].

Corollary 1 (*surjective_pri*).

Corollary 2 (*nonempty_product* and variants).

Corollary 3 (*productb_monotone1*, *productb_monotone2*).

Proposition 7 (*bijective_pam*) states associativity of the product, [100].

Proposition 8 (*distrib_union_inter* and *distrib_inter_union*) states distributivity of union over intersection and intersection over union, [101].

Corollary (*distrib_union2_inter* and *distrib_inter2_union*).

Proposition 9 (*distrib_prod_union* and *distrib_prod_intersection*) states distributivity of product over union and intersection, [103].

Corollary 1 (*partition_product*).

Corollary 2 (*distrib_prod2_union* and *distrib_prod2_intersection*) .

Proposition 10 (*distrib_inter_prod* and *distrib_prod_intersection*) says that the intersection of a product is the product of the intersection, [104].

Corollary (*distrib_prod_inter2_prod* and *distrib_inter_prod_inter*).

Proposition 11 says that composition of extensions is extension of compositions.

Corollary (*injective_ext_map_prod* and *injective_ext_map_prod*).

Section 6

Proposition 1 (*equivalence_cor_pr*) says: « A correspondence Γ between X and X is an equivalence on X if and only if it satisfies the following conditions: (a) X is the domain of Γ; (b) $\Gamma = \Gamma^{-1}$; (c) $\Gamma \circ \Gamma = \Gamma$ », [114].

Criterion C55 (*related_e_rw*) characterizes the canonical projection, [117].

Criterion C56 (*rel_on_quo_pr*) « Let R$\{x, x'\}$ be an equivalence relation on a set E and let P$\{x\}$ be a relation that does not contain the letter x' and is compatible (with respect to x) with the equivalence relation $\{x, x'\}$. Then, if t does not appear in P$\{x\}$, the relation "$t \in$ E/R and $(\exists x)(x \in t$ and P$\{x\}$)" is equivalent to the relation "$t \in$ E/R and $(\forall x)(x \in t$ and P$\{x\}$)" ». [120].

Criterion C57 *exists_unique_fun_on_quotient*) « Let R be an equivalence relation on a set E, and let g be the canonical mapping of E onto E/R. Then a mapping f of E into F is compatible with R is and only if f can be put in the form $h \circ g$, where h is a mapping of E/R into F. The mapping h is uniquely defined by f; if f is any section of g, we have $h = f \circ s$. »[123]

9.7 Notations and Definitions

In many cases we indicate the page on which an object is defined.

Symbols

$x \wedge y$ is often replaced by "and". The Coq equivalent is `/\`.

$x \vee y$ is often replaced by "or". The Coq equivalent is `\/`.

$\neg x$ is often replaced by "not". The Coq equivalent is `~`.

$\square$ is a dummy variable for Bourbaki, [6].

R$\{x\}$ is a Bourbaki notation, meaning that R is a relation that may depend on x. If R is a relation that depends on y, it is also $(x|y)$R.

τ_x(R) is a Bourbaki notation, it is the generic element satisfying R$\{x\}$, [10].

$x \Longrightarrow y$ is represented in Coq by `x -> y`.

$x \mapsto y$ is represented in Coq by `fun x => y`.

$x \to y$ is a Coq notation meaning the type of functions from type x to type y.

$x = y$ is equality. We use it as synonym to $\iff$.

$(a|b)c$ is a Bourbaki notation, meaning the relation obtained by replacing b by a in c, [7].

$x : y$ is a Coq notation meaning that x is of type y.

$f(x)$ is the value of the function f at point x, parentheses are sometimes omitted.

$f\langle x\rangle$ is the value of f on the set x, see *fun_image*, *image_by_graph*, *image_by_fun*.

$\overset{-1}{f}\langle x\rangle$, see *inverse_image*.

$(\forall x)$P and *forall x, p* are similar constructions, [10].

$(\exists x)$P and *exists x, p* are similar constructions, [10].

$(\exists! x)$P means sometimes *exists_unique*.

$x \in y$, $x \ni y$ (is element of): see *inc* and *elt*.

$x \subset y$ (is subset of): see *sub*.

$\emptyset$ (empty set): see *emptyset*.

$\{x, \mathrm{R}\}$ (set of x such that R): see *Zo*.

$\{x\}$, $\{x, y\}$: see *singleton* or *doubleton*.

$a - b$, $a \setminus b$, $\complement a$: see *complement*.

(x, y) (ordered pair): see *J*.

$\bigcup \mathrm{X}$, $\bigcup_{\iota\in\mathrm{I}} \mathrm{X}_\iota$, see *union*.

$a \cup b$, $a \cap b$, see *union2*, *intersection2*.

$\mathrm{A} \times \mathrm{B}$, $u \times v$, $\mathrm{R} \times \mathrm{R}'$, see *product*, *ext_to_prod*, *prod_of_relation*.

$f \circ g$, see *fcompose*, *gcompose*, *compose_graph*, *compose*, *composeC*.

Δ_A, see *diagonal*.

$\overset{-1}{\mathrm{G}}$ see *inverse_graph* , *inverse_fun* or *inverseC*.

$x \mapsto y$ or $x \to y$ is the function that maps x to y, for instance $x \mapsto \sin(x)$ (source and target are implicit).

$\boldsymbol{x \to T}$ $(\boldsymbol{x \in A, T \in C})$, is the function with source A, target C that maps x to T, [58].

$(f_x)_{x\in\mathrm{A}}$ is a shorthand for $x \to f(x)$ $(x \in \mathrm{A})$; see above, the piece $\boldsymbol{T \in C}$ is implicit.

$\hat{f}$, see *extension_to_parts*

F^E, see *set_of_gfunctions*.

$\mathscr{F}(\mathrm{E};\mathrm{F})$ see *set_of_functions*.

$\Phi(\mathrm{E},\mathrm{F})$ see *set_of_sub_functions*.

f_x, f_y sometimes denotes the mappings $y \mapsto f((x, y))$ or $x \mapsto f((x, y))$, implemented as *first_partial_fun*, *second_partial_fun*, [92].

$\tilde{f}$, sometimes denotes the mappings $x \mapsto f_x$ or $y \mapsto f_y$. Implemented as *first_partial_function*, *second_partial_function*, [92].

$f \mapsto \tilde{f}$, implemented as *first_partial_map*, *second_partial_map*, is a bijection from $\mathscr{F}(\mathrm{B} \times \mathrm{C};\mathrm{A})$ into $\mathscr{F}(\mathrm{B};\mathscr{F}(\mathrm{C};\mathrm{A})$ or $\mathscr{F}(\mathrm{C};\mathscr{F}(\mathrm{B};\mathrm{A}))$, [92].

$\prod_{\iota\in\mathrm{I}} \mathrm{X}_\iota$ see *productt*.

$(x_\iota)_{\iota\in\mathrm{I}}$ denotes an element of a product indexed by I.

$x \overset{r}{\sim} y$ is sometimes used instead of $r(x, y)$ or $(x, y) \in r$, especially when r is the graph of an equivalence relation.

$g_\mathrm{E}(\sim)$, the graph of $\sim$ on E, see *graph_on*.

$\sim_f$ may denote *eq_rel_associated f*.

$\bar{x}$, may denote the equivalence class of x, see *class*.

$\hat{x}$ may denote a representative of the equivalence class x.

E/ $\sim$, E/R, see *quotient*.

R/S see *quotient_of_relations*.

X_f sometimes means $f^{-1}\langle f\langle \mathrm{X}\rangle\rangle$, see *inverse_direct_value*.

R_A see *induced_relation*.

¶ is not defined. We use it as a paragraph separator.

Letters

$\mathscr{B}$ see *Bo*.

$\mathscr{C}_\mathrm{C}(a,b)$, $\mathscr{C}_\mathrm{T}(p,q)$, $\mathscr{C}(p)$: see *by_cases a b*, *chooseT* and *choose*.

$\mathrm{C}_{xy}a$ stands for *constant_function x y a*, it is the constant function from x to y with value a, [53].

$\mathrm{C}_\mathrm{R}x$ may denote the equivalence class of x for R, see *class*.

$\mathrm{Coll}_{\boldsymbol{x}}\boldsymbol{R}$ says that R is collectivizing in x, [13].

$\mathscr{E}$, see *Set*.

$\mathscr{E}_x(\mathrm{R})$ appears in the English version where $\{x,\mathrm{R}\}$ is used in the French version; see *Zo*.

I_A, see *identity*.

I_{xy} see *inclusionC*, *canonical_injection*.

$\mathscr{L}_\mathrm{X}f$, $\mathscr{L}f$, $\mathscr{L}_{\mathrm{A;B}}f$ (creating functions): see *L*, *acreate*, *BL*.

$\mathscr{M}f$, $\mathscr{M}_{\mathrm{A;B}}f$ (inverse of $\mathscr{L}$), see *bcreate1* and *bcreate*.

$\mathfrak{P}(x)$, see *powerset*.

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$, $\mathrm{pr}_\iota f$, $\mathrm{pr}_\mathrm{J} f$ (projections), see P, Q, *pr_i*, *pr_j*.

$\mathscr{R}x$ see *Ro*.

$\mathrm{R}_{ab}f$ (restriction) see [57].

$\mathscr{V}(x,f)$, $\mathscr{V}_f x$ (value of a function): see *V*.

$\mathscr{W}_f x$ (value of a function): see *W*.

$\mathscr{X}(f,y)$, see *Xo*.

$\mathscr{Y}(\mathrm{P},x,y)$ see *Yo*.

$\mathscr{Z}(x,\mathrm{P})$ see *Zo*.

Words

acreate f, $\mathscr{L}f$, is the correspondence associated to the Coq function f, [44].

agrees_on x f f', *agreeC x f f'* is the property that for all $a \in x$, $f(a)$ and $f'(a)$ are defined and equal, [55].

bcreate f A B, $\mathscr{M}_{\mathrm{A;B}}f$, is a kind of inverse of $\mathscr{L}$, [51].

bcreate1 f, $\mathscr{M}f$, is a kind of inverse of $\mathscr{L}$ [51].

bijective f, *bijectiveC f*, means that f is a bijection, [60].

BL f a b, $\mathscr{L}_{\mathrm{A;B}}f$, *fun_function f a b*, is function from A to B whose graph is $\mathscr{L}_\mathrm{A}f$, [58].

Set or $\mathscr{E}$ is the type of sets, [13].

Bo, $\mathscr{B}$, is an inverse of $\mathscr{R}$, [18].

by_cases a b, $\mathscr{C}_\mathrm{C}(a,b)$, defines an object by applying a if P is true, and b if P is false, [18].

canonical_injection x y, I_{xy}, is the inclusion map on $x \subset y$, , [62].

canon_proj r, is the mapping $x \mapsto \bar{x}$ from E onto E/R, the quotient set of r, [116].

class r x is the class of x for the equivalence relation r, [115].

choose p, $\mathscr{C}(p)$, is some x such that $p(x)$ is true, the empty set if no x satisfies p, [19].

chooseT p q, $\mathscr{C}_{\mathrm{T}}(p,q)$, is our basic axiom of choice, [15].

coarse x is $x \times x$, [112].

coarser_covering I f J g, *coarser_c f g*, two definitions that say for all $j \in \mathrm{J}$ there is $i \in \mathrm{I}$ such that $g_j \subset f_i$ or for all $g_j \in g$ there is $f_i \in f$ such that $g_j \subset f_i$, [82].

compatible_with_equiv_p p r means that $p(x)$ and $x \overset{r}{\sim} y$ implies $p(y)$, [119].

compatible_with_equiv f r means that $x \overset{r}{\sim} y$ is equivalent to $f(x) = f(y)$, [122].

compatible_with_equivs f r r' means that $x \overset{r}{\sim} y$ is equivalent to $f(x) \overset{r'}{\sim} f(y)$, [122].

complement a b, $a - b$, $a \setminus b$, $\complement b$, is the set of element of a not in b, [23].

composableC f g, *composable f g* is the condition on correspondences (resp. functions) f and g for $f \circ g$ to be a correspondence (resp. function), [48], [59].

compose_graph f g, $f \circ g$, composition of two graphs, [47].

compose f g, *composeC f g*, $f \circ g$, is the composition of two functions, [48], [54].

constant_graph s x is the graph of the constant function with domain s and value x, [97].

correspondenceC is a data type with three slots, source, target and graph, [43]

corr_value f associates to a correspondence f its triple (G, A, B), [43].

covering f x, *covering_f I f x*, *covering_s f x*, three variants of a family of sets (defined by f and I) whose union contains x, [82].

cut x p is the set of all x that satisfy p, [21].

cut r x is $r\langle\{x\}\rangle$, replaced by *im_singleton* [45].

diagonal A, Δ_{A}, is the set of all (x,x) such that $x \in \mathrm{A}$, [42].

diagonal_application A is the diagonal mapping $x \mapsto (x,x)$ of A into Δ_{A}, [62].

diagonal_graphp I E is the set of graphs of constant functions from I to E, [97].

disjoint x y means $x \cap y = \emptyset$, [84].

disjoint_union f, *disjoint_union_fam f* are two variants of the disjoint union of the family of sets f, [87].

domain f is the set of x for which there is an y with $(x,y) \in f$, it is $\mathrm{pr}_1\langle f\rangle$, [31].

doubleton x y, $\{x,y\}$, is a set with elements x and y, [22].

EEE is a shorthand for the type *Set* → *Set* → *Set*.

EEP is a shorthand for the type *Set* → *Set* → *Prop*.

elt x y, $x \ni y$, is the same as $y \in x$, [17].

empty_function, *empty_functionC* is the identity on $\emptyset$, [52].

emptyset, $\emptyset$, is a set without elements, [17].

eq_rel_associated f is the graph of the equivalence relation $f(x) = f(y)$, [114].

equipotent x y means that there is a bijection from x to y.

equivalence_associated f is the equivalence relation $f(x) = f(y)$, [114].

equivalence_r r, *equivalence_re r x*, says that the relation r is an equivalence relation (in x), [109].

equivalence_corr r says that the correspondence r is associated to an equivalence, [112].

exists_unique p,$(\exists! x)\,p$, (this notation is not in Bourbaki) means that there exists a unique x such that $p(x)$, [17].

extends g f, *extendsC g f* says $g(x) = f(x)$ whenever $f(x)$ is defined, [56].

ext_map_prod I X Y g is the function $(x_\iota)_{\iota\in I} \mapsto (g_\iota(x_\iota))_{\iota\in I}$ from $\prod_I X_\iota$ into $\prod_I Y_\iota$, [106].

ext_to_prod u v is the function $(x, y) \mapsto (u(x), v(y))$, sometimes denoted $u \times v$, [72]

extension_to_parts f, denotes the function $x \mapsto f\langle x\rangle$, from $\mathfrak{P}(A)$ to $\mathfrak{P}(B)$, [89]

finer_equivalence s r, comparison of equivalences, $x \overset{s}{\sim} y$ implies $x \overset{r}{\sim} y$, [128].

first_proj g is the function $x \mapsto \mathrm{pr}_1 x\ (x \in g)$.

first_proj_equiv x y, *first_proj_equivalence x y*, is the equivalence associated to *first_proj* on the set $x \times y$, [117].

fcompose f g, $f \circ g$, composition of two graphs, without assumption, [33].

fcomposable f g says that graphs g and $f \circ g$ have the same domain, [33].

fgraph f says that f is a functional graph, [31].

functional_graph f says that f is a functional graph, [48].

fun_image x f, $f\langle x\rangle$, is the value of f on the set x, [25].

fun_on_quotient r f, *function_on_quotient r f b*, *function_on_quotients*, *fun_on_quotients r r' f*, the function obtained from f on passing to the quotient of r (or r and r'), [123], [123].

fun_set_to_prod E X is the canonical bijection between $(\prod X_\iota)^E$ and $\prod X_\iota^E$, [107].

function_prop f s t, *function_prop_sub f s t*. This is the property that f is a function from s into t, or into a subset of t, [84].

gcompose f g, $f \circ g$, composition of two graphs, assumes that range g is a subset of domain f, [33].

graph f is a part of a correspondence, [43].

graph_on r X is the graph of the relation r restricted to X, [111].

identity A, I_A, is is the graph of the identity function on the set A, [34].

identity_fun A, I_A, is the identity function on the set A, [48].

IM stands for the image of a function. Its axioms implement the Scheme of Selection and Union, [16].

image_by_fun f A, $f\langle A\rangle$, is $\{t, \exists x \in A, t = f(x)\}$, [44].

image_by_graph f A, $f\langle A\rangle$ is $\{t, \exists x \in A, (x, t) \in f\}$, [44].

image_of_fun f, is the image of f, [44].

inc x y or $x \in y$ means that x is an element of y, [13].

inclusionC x y, I_{xy}, is the inclusion map on $x \subset y$ as a Coq function, [54].

induced_relation R A, R_A, is the equivalence induced by R on A, [127].

injective f, *injectiveC f*, means that f is an injection, [60].

in_same_coset f is the relation "there exists i such that $x \in f(i)$ and $y \in f(i)$" between x and y, [118].

intersection X, $\bigcap X$, is the intersection of a set of sets, [27].

intersectiont I f, *intersectionf x f*, *intersectiont g*, $\bigcap_{\iota\in I} X_\iota$ is the set of elements a such that forall $\iota \in I$ we have $a \in X_\iota$, [76].

intersection2 X Y, $X \cap Y$, is the intersection of two sets [27].

intersection_covering, intersection of coverings, [82].

inverse_direct_value f X, X_f, is $f^{-1}\langle f\langle X\rangle\rangle$, [120].

inverse_graph G, $\overset{-1}{\mathrm{G}}$, inverse graph of the graph G, [45].

inverse_fun f or *inverseC a b f H*, $\overset{-1}{f}$, inverse of the function f, [46], [63].

inverse_image x f, $\overset{-1}{f}\langle x\rangle$, is the inverse value of f on the set x, [32].

inv_image_relation f r, is the inverse image of the relation r under the function f, [127].

inv_image_by_graph f x, *inv_image_by_fun r x*, $\overset{-1}{f}\langle x\rangle$, direct image of a set by the inverse function, [46]

inv_corr_value t associates to a $t = (\mathrm{G},\mathrm{A},\mathrm{B})$ its correspondence f, [43].

inv_graph_canon G is the bijection $(x,y) \mapsto (y,x)$ from G to G^{-1}, [63].

is_class r x says that x is an equivalence class for r, [115]

is_correspondence f says that f is associated to a triple $(\mathrm{G},\mathrm{A},\mathrm{B})$, [43]

is_equivalence r says that the graph r is an equivalence, [110].

is_function f says that f is a function in the sense of Bourbaki, [49].

is_graph f says that f is a set of pairs, [31].

is_graph_of g r is true if g is the graph of the relation r, [111].

is_left_inverse r f means that r is a retraction or left-inverse of f, and $r \circ f$ is the identity, [65].

is_reflexive r says that the graph r is reflexive, [110].

is_restriction f g says that f is the restriction of g to some set, [31]

is_right_inverse s f means that s is a section or right-inverse of f, and $f \circ s$ is the identity, [65].

is_singleton x means that x is a singleton.

is_symmetric r says that the graph r is symmetric, [110].

is_transitive r says that the graph r is transitive, [110].

J x y, or (x,y), is an ordered pair, formed of two items x and y, [24].

L X f, *fcreate X f*, $\mathscr{L}_{\mathrm{X}} f$ is the graph formed of all $(x, f(x))$ with $x \in \mathrm{X}$, [33].

largest_partition x is the set of all singletons of x.

left_inverseC, left inverse of a Coq function, [66].

LHS is the left hand side of an equality.

Lvariant a b x y, *variant a x y*, *Lvariantc x y*, these are functions whose range is the doubleton $\{x,y\}$, [85].

mutually_disjoint f says that for all distinct i and j, $f(i)$ and $f(j)$ are disjoint, [84]

$x \neq y$, *neq x y*, *x <> y* is inequality, [17].

one_point is the basic singleton, [21].

P z, $\mathrm{pr}_1 z$ denotes x if z is the pair (x,y), [24].

partial_fun1 f y, *partial_fun1 f x*, partial functions, [71].

partition y x, *partition_s y x*,*partition_fam f x*, thee variants that say that y or f is a partition of x, [84].

partition_relation f x is the equivalence relation associated to the partition f of x, [118].

partition_with_complement X A, is the partition of X formed of A and its complementary set, [85].

powerset x, $\mathfrak{P}(x)$, is the set of subsets of x, [25].

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$ stand for *pr1 z* and *pr2 z*. These are also denoted by P and Q. If z is the pair (x,y), these functions return x and y respectively, [24].

pr_i f i, *pr_it f i*, $\mathrm{pr}_i f$, denotes a component of an element of a product. [95].

pr_j f J, $\mathrm{pr}_J f$, is the function $(x_\iota)_{\iota\in I} \mapsto (x_\iota)_{\iota\in J}$, [99].

prod_assoc_map is the function whose bijectivity is the "theorem of associativity of products", [100].

prod_of_function u v, is the function $x \mapsto (u(x), v(x))$, [105].

prod_of_products_canon F F', is the bijection between $\prod \mathrm{F}_\iota \times \prod \mathrm{F}'_\iota$ and $\prod(\mathrm{F}_\iota \times \mathrm{F}'_\iota)$, [105].

prod_of_relation R R', $\mathrm{R} \times \mathrm{R}'$, is the product of two equivalences, [130].

product A B, $\mathrm{A} \times \mathrm{B}$, is the set of all pairs (a, b) with $a \in \mathrm{A}$ and $b \in \mathrm{B}$, [29]. See also *ext_to_prod u v*.

producttt I X, *product b g* or *productf I f*, $\prod_{\iota\in I} \mathrm{X}_\iota$ is the product of a family of sets, [95].

product1 x a is the product of the family defined on the singleton $\{a\}$ via value x, [96].

product1_canon x a is the canonical application from x into *product1 x a*, [96].

product2 x y is the product of the family defined on the doubleton $\{a, b\}$ via value x and y, [97].

product2_canon x y is the canonical application from $x \times y$ into *product2 x y*, [97].

product_compose, auxiliary function used for change of variables in a product, [98].

Q z, $\mathrm{pr}_2 z$ denotes y if z is the pair (x, y), [24].

quotient R, E/R, is the set of equivalence classes of R, [115]

quotient_of_relations r s, R/S, is the quotient of two equivalences, [129]

range f is the set of y for which there is an x with $(x, y) \in f$, it is $\mathrm{pr}_2\langle f\rangle$, [31].

reflexive_r r x says that the relation r is reflexive in x, [109].

related r x y is a shot-hand for $(x, y) \in r$, [41].

relation_on_quotient p r is the relation induced by $p(x)$ on passing to the quotient (with respect to x) with respect to R, [119]

rep x is an element y such that $y \in x$, whenever x is not empty, [19].

representative_system s f x means that, for all i, $s \cap \mathrm{X}_i$ is a singleton, where X_i is a partition of x associated to the function f, [119].

representative_system_function g f x, means that g is an injection whose image is a system of representatives (see definition above), [119].

restr x G is the restriction to x of the graph G, [34].

restricted_eq E is the relation "$x \in$ E and $y \in$ E and $x = y$", [112].

restriction_function f x is like *restr*, but f and the restrictions are functions, [55].

restriction2_axioms f x y is the condition: f is a function whose source contains x, whose target contains y, moreover $a \in x$ implies $f(a) \in y$, [57].

restriction2 f x y, *restriction2C f x y*, restriction of f as a function $x \to y$, [57].

restrictionC f H is the restriction to x of the function $f : a \to b$, where H proves $x \subset a$ implicitly, [55].

restriction_product f j is the product of the restrictions of $\prod f$ to J, [99].

restriction_to_image f is the restriction of the Coq function f to its range, [74].

retraction: see *is_left_inverse*.

RHS is the right hand side of an equality.

right_inverseC, right inverse of a Coq function, [66].

Ro x or $\mathscr{R}x$ converts its argument x of type u to a set, which is an element of u, [15].

saturated r x means: for every $y \in x$, the class of x for the relation r is a subset of x, [120].

saturation_of r x is the saturation of x for r, [121].

second_proj g is the function $x \mapsto \mathrm{pr}_2 x$ $(x \in g)$.

section: see *is_right_inverse*.

section_canon_proj R is the function from E/R into E induced by *rep*, [122].

set_of_correspondences A B means the set of triples associated to correspondences from A to B, it is $\mathfrak{P}(\mathrm{A} \times \mathrm{B}) \times \{\mathrm{A}\} \times \{\mathrm{B}\}$, [44].

set_of_endomorphisms E, is the set of triples (G, E, E) associated to functions from E into E, [90]

set_of_functions E F, denoted $\mathscr{F}(\mathrm{E};\mathrm{F})$, is the set of triples (G, E, F) associated to functions from E into F, [90]

set_of_gfunctions E F, denoted F^{E}, is the set of graphs of functions from E to F, [90]

set_of_sub_functions E F, denoted $\Phi(\mathrm{E};\mathrm{F})$ is the set of triples (G, A, F) associated to functions from $\mathrm{A} \subset \mathrm{E}$ into F, [90]

singleton x, $\{x\}$, is a set with one element, [21].

sof_value x y z converts three elements into a correspondence, [90].

small_set x means that x has at most one element, [53].

smallest_partition x is the singleton $\{x\}$.

source f contains (resp. is equal to) the domain of the graph of a correspondence f (resp. function f) [43], [49].

strict_sub x y, $x \subsetneq y$, means $x \subset y$ and $x \neq y$, [17].

sub x y, $x \subset y$, means that x is a subset of y, [13].

surjective f, *surjectiveC f*, means that f is a surjection, [60].

substrate r is the union of the domain and range [109].

symmetric_r r says that the relation r is symmetric, [109].

target f contains the range of the graph of a correspondence f, [43].

transf_axioms f A B says that for all $x \in \mathrm{A}$ we have $f(x) \in \mathrm{B}$, case where $\mathscr{L}_{\mathrm{A};\mathrm{B}} f$ is a function, [58].

transitive_r r says that the relation r is transitive, [109].

two_points is the basic doubleton, [22].

union X, $\bigcup \mathrm{X}$, is the union of a set of sets, [26],

uniont I f, *unionf x f*, *uniont g*, $\bigcup_{\iota \in \mathrm{I}} \mathrm{X}_\iota$ is the set of elements a such that $a \in \mathrm{X}_\iota$ for some $\iota \in \mathrm{I}$, [76].

union2 a b, $a \cup b$, is the union of two sets, [26].

V x f, $\mathscr{V}(x, f)$ or $\mathscr{V}_f x$, is the value at the point x of the graph f, [25].

variant, see *Lvariant*.

W x f, $\mathscr{W}_f x$, is the value at the point x of the function f, [49].

Xo f y, $\mathscr{X}(f, y)$, this is $f(x)$ if $y = \mathscr{R} x$, [20].

Yo P x y, $\mathscr{Y}(\mathrm{P}, x, y)$, is a function that associates to z the value x is P is true, and y if P is false, [19].

Zo x R, $\mathscr{Z}(x, \mathrm{R})$, $\mathscr{E}_x(\mathrm{R})$ or $\{x, \mathrm{R}\}$: it is the set of all x that satisfy R, [13] [20].

Index

Bibliography

[1] Yves Bertod and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[2] N. Bourbaki. *Elements of Mathematics, Theory of Sets.* Springer, 1968.

[3] N. Bourbaki. *Éléments de mathématiques, Théorie des ensembles.* Diffusion CCLS, 1970.

[4] Coq Development Team. The Coq reference manual. http://coq.inria.fr.

[5] Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1979.

[6] Jean-Louis Krivine. *Théorie axiomatique des ensembles.* Presses Universitaires de France, 1972.

[7] Edward Nelson. Internal set theory: a new approach to nonstandard analysis. *Bulletin of the American Mathematical Society*, 1977.

Contents

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399